

## FFmpeg的HEVC解码器源代码简单分析：解析器（Parser）部分

2015年06月09日 23:19:48 阅读数：13199

HEVC源代码分析文章列表：

【解码 -libavcodec HEVC 解码器】

[FFmpeg的HEVC解码器源代码简单分析：概述](#)

[FFmpeg的HEVC解码器源代码简单分析：解析器（Parser）部分](#)

[FFmpeg的HEVC解码器源代码简单分析：解码器主干部分](#)

[FFmpeg的HEVC解码器源代码简单分析：CTU解码（CTU Decode）部分-PU](#)

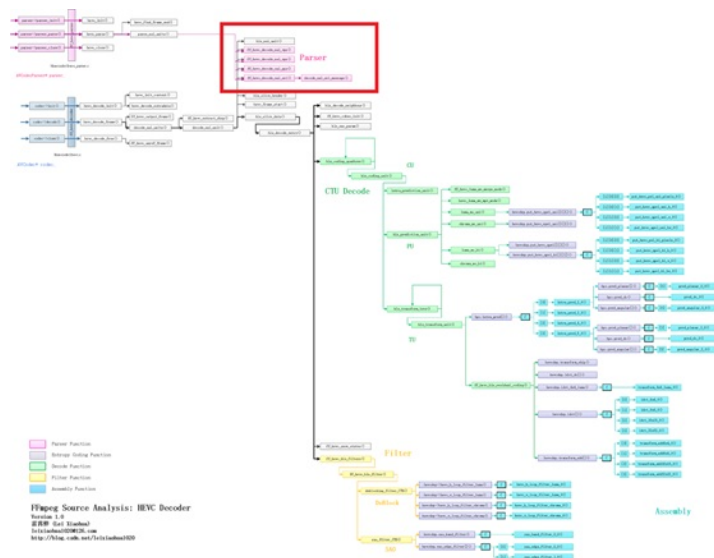
[FFmpeg的HEVC解码器源代码简单分析：CTU解码（CTU Decode）部分-TU](#)

[FFmpeg的HEVC解码器源代码简单分析：环路滤波（LoopFilter）](#)

上篇文章概述了FFmpeg的libavcodec中HEVC（H.265）解码器的结构；从这篇文章开始，具体研究HEVC解码器的源代码。本文分析HEVC解码器中解析器（Parser）部分的源代码。这部分的代码用于分割HEVC的NALU，并且解析SPS、PPS、SEI等信息。解析HEVC码流（对应AVCodecParser结构体中的函数）和解码HEVC码流（对应AVCodec结构体中的函数）的时候都会调用该部分的代码完成相应的功能。

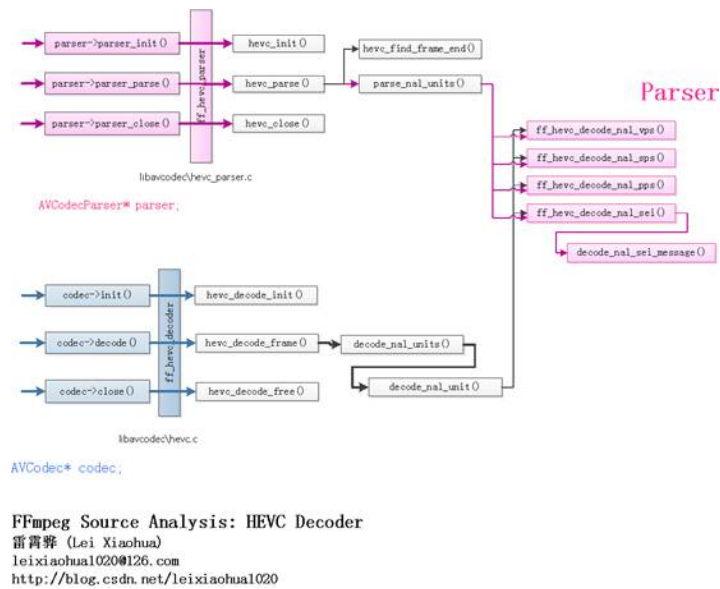
## 函数调用关系图

FFmpeg HEVC解析器（Parser）部分在整个HEVC解码器中的位置如下图所示。



[单击查看更清晰的大图](#)

HEVC解析器（Parser）部分的源代码的调用关系如下图所示。



[单击查看更清晰的大图](#)

从图中可以看出，HEVC解析器调用了`parse_nal_units()`，HEVC解码器调用了`decode_nal_units()`，而上述两个函数都调用了下面几个解析函数：

`ff_hevc_decode_nal_vps()`：解析VPS。

`ff_hevc_decode_nal_sps()`：解析SPS。

`ff_hevc_decode_nal_pps()`：解析PPS。

`ff_hevc_decode_nal_sei()`：解析SEI。

下文将会分别这几个函数。

## ff\_hevc\_decoder

`ff_hevc_decoder`是HEVC解码器对应的`AVCodec`结构体。该结构体的定义位于`libavcodec/hevc.c`，如下所示。

```
[cpp]
1. AVCodec ff_hevc_decoder = {
2.     .name = "hevc",
3.     .long_name = NULL_IF_CONFIG_SMALL("HEVC (High Efficiency Video Coding)"),
4.     .type = AVMEDIA_TYPE_VIDEO,
5.     .id = AV_CODEC_ID_HEVC,
6.     .priv_data_size = sizeof(HEVCContext),
7.     .priv_class = &hevc_decoder_class,
8.     .init = hevc_decode_init,
9.     .close = hevc_decode_free,
10.    .decode = hevc_decode_frame,
11.    .flush = hevc_decode_flush,
12.    .update_thread_context = hevc_update_thread_context,
13.    .init_thread_copy = hevc_init_thread_copy,
14.    .capabilities = CODEC_CAP_DR1 | CODEC_CAP_DELAY |
15.                  CODEC_CAP_SLICE_THREADS | CODEC_CAP_FRAME_THREADS,
16.    .profiles = NULL_IF_CONFIG_SMALL(profiles),
17. };
```

从源代码中可以看出，HEVC解码器的解码函数是`hevc_decode_frame()`。由于本文主要分析HEVC解析器，所以不对解码函数进行分析。在这里只需要知道`hevc_decode_frame()`调用了`decode_nal_units()`，而`decode_nal_units()`最终调用了`ff_hevc_decode_nal_sps()`等解析函数即可。

## ff\_hevc\_parser

`ff_hevc_parser`是HEVC解析器对应的`AVCodecParser`结构体。该结构体的定义位于`libavcodec/hevc_parser.c`，如下所示。

```
[cpp]
1. AVCodecParser ff_hevc_parser = {
2.     .codec_ids = { AV_CODEC_ID_HEVC },
3.     .priv_data_size = sizeof(HEVCParserContext),
4.     .parser_init = hevc_init,
5.     .parser_parse = hevc_parse,
6.     .parser_close = hevc_close,
7.     .split = hevc_split,
8. };
```

从源代码中可以看出，HEVC解析器的初始化函数是`hevc_init()`，解析函数是`hevc_parse()`，关闭函数是`hevc_close()`。

## hevc\_init()

`hevc_init()`是HEVC解析器的初始化函数，该函数的定义如下所示。

```
[cpp]
1. static int hevc_init(AVCodecParserContext *s)
2. {
3.     HEVCContext *h = &((HEVCParserContext *)s->priv_data)->h;
4.     h->HEVCIdc = av_mallocz(sizeof(HEVCLocalContext));
5.     h->skipped_bytes_pos_size = INT_MAX;
6.
7.     return 0;
8. }
```

可以看出hevc\_init()简单地给内部成员变量分配了内存。

## hevc\_close()

hevc\_close()是HEVC解析器的关闭函数，该函数的定义如下所示。

```
[cpp]
1. static void hevc_close(AVCodecParserContext *s)
2. {
3.     int i;
4.     HEVCContext *h = &((HEVCParserContext *)s->priv_data)->h;
5.     ParserContext *pc = &((HEVCParserContext *)s->priv_data)->pc;
6.
7.     av_freep(&h->skipped_bytes_pos);
8.     av_freep(&h->HEVCIdc);
9.     av_freep(&pc->buffer);
10.
11.     for (i = 0; i < FF_ARRAY_ELEMS(h->vps_list); i++)
12.         av_buffer_unref(&h->vps_list[i]);
13.     for (i = 0; i < FF_ARRAY_ELEMS(h->sps_list); i++)
14.         av_buffer_unref(&h->sps_list[i]);
15.     for (i = 0; i < FF_ARRAY_ELEMS(h->pps_list); i++)
16.         av_buffer_unref(&h->pps_list[i]);
17.
18.     av_buffer_unref(&h->current_sps);
19.     h->sps = NULL;
20.
21.     for (i = 0; i < h->nals_allocated; i++)
22.         av_freep(&h->nals[i].rbsp_buffer);
23.     av_freep(&h->nals);
24.     h->nals_allocated = 0;
25. }
```

可以看出hevc\_close()释放了内部成员变量的内存。

## hevc\_parse()

hevc\_parse()是HEVC解析器中最关键的解析函数。该函数的定义如下所示。

```

1.  /*
2.  * 解析码流
3.  *
4.  * 注释：雷霄骅
5.  * leixiaohua1020@126.com
6.  * http://blog.csdn.net/leixiaohua1020
7.  */
8.  static int hevc_parse(AVCodecParserContext *s,
9.                      AVCodecContext *avctx,
10.                     const uint8_t **poutbuf, int *poutbuf_size,
11.                     const uint8_t *buf, int buf_size)
12.  {
13.      int next;
14.      ParseContext *pc = &((HEVCParserContext *)s->priv_data)->pc;
15.      //PARSER_FLAG_COMPLETE_FRAMES为1的时候说明传入的就是完整的1帧数据
16.      //这时候不用再分割NALU
17.      //PARSER_FLAG_COMPLETE_FRAMES为0的时候说明传入的是任意一段数据
18.      //需要先分离出完整的NALU
19.      if (s->flags & PARSER_FLAG_COMPLETE_FRAMES) {
20.          next = buf_size;
21.      } else {
22.          //分割NALU
23.          //通过查找起始码0x000001的方法
24.          next = hevc_find_frame_end(s, buf, buf_size);
25.          //合并
26.          if (ff_combine_frame(pc, next, &buf, &buf_size) < 0) {
27.              *poutbuf = NULL;
28.              *poutbuf_size = 0;
29.              return buf_size;
30.          }
31.      }
32.      //解析NALU内容（不解码）
33.      parse_nal_units(s, avctx, buf, buf_size);
34.
35.      *poutbuf = buf;
36.      *poutbuf_size = buf_size;
37.      return next;
38.  }

```

从源代码可以看出，hevc\_parse()主要做了两步工作：

- (1) 判断传入的flags 中是否包含PARSER\_FLAG\_COMPLETE\_FRAMES。如果包含，则说明传入的是完整的一帧数据，不作任何处理；如果不包含，则说明传入的不是完整的一帧数据而是任意一段HEVC数据，则需要调用hevc\_find\_frame\_end()通过查找“起始码”（0x00000001或者0x000001）的方法，分离出完整的一帧数据。
- (2) 调用parse\_nal\_units()完成了NALU的解析工作。

下面分别看一下这两步中的两个函数。

## hevc\_find\_frame\_end()

hevc\_find\_frame\_end()用于从HEVC码流中分离出完整的NALU。该函数的定义位于libavcodec\hevc\_parser.c，如下所示。

```

1.  /**
2.   * Find the end of the current frame in the bitstream.
3.   * @return the position of the first byte of the next frame, or END_NOT_FOUND
4.   */
5.  //分割NALU
6.  static int hevc_find_frame_end(AVCodecParserContext *s, const uint8_t *buf,
7.                                int buf_size)
8.  {
9.      int i;
10.     ParseContext *pc = &((HEVCParserContext *)s->priv_data)->pc;
11.     //一个一个字节进行处理
12.     for (i = 0; i < buf_size; i++) {
13.         int nut;
14.         //state64可以存8个字节
15.         //buf[i]存入state64
16.         pc->state64 = (pc->state64 << 8) | buf[i];
17.
18.         //起始码定义#define START_CODE 0x000001
19.         //state64右移24bit之后, 再对比是否为起始码0x000001
20.         if (((pc->state64 >> 3 * 8) & 0xFFFFF) != START_CODE)
21.             continue;
22.         //找到起始码之后
23.         /*
24.          * 此时state64内容如下:
25.          *
26.          * |-----|-----|-----|-----|-----|-----|-----|
27.          *
28.          * | buf | buf | buf | buf | buf | buf | buf |
29.          * | [t-5] | [t-4] | [t-3] | [t-2] | [t-1] | [t] |
30.          *
31.          * Start Code:
32.          * 0x000001
33.          *
34.          * NALU Header:
35.          * forbidden_zero_bit: 1bit. 取值0。
36.          * nal_unit_type: 6 bit. NALU类型。
37.          * nuh_layer_id: 6 bit. 目前取值为0 (保留以后使用) .
38.          * nuh_temporal_id_plus1: 3 bit. 减1后为NALU时域层标识号TemporalID。
39.          *
40.          */
41.         //state64右移16bit之后, state64最低字节为起始码后面的1Byte。即为NALU Header的前一个字节
42.         //NALU Header的前一个字节中, 第1bit为forbidden_zero_bit, 取值为0;
43.         //2-7bit为nal_unit_type; 第8bit为nuh_layer_id, 取值为0。
44.
45.         //在这里state64右移(16+1)bit, 然后相与0x3F(00111111)
46.         //即得到了nal_unit_type
47.         nut = (pc->state64 >> 2 * 8 + 1) & 0x3F;
48.
49.         // Beginning of access unit
50.         if ((nut >= NAL_VPS && nut <= NAL_AUD) || nut == NAL_SEI_PREFIX ||
51.             (nut >= 41 && nut <= 44) || (nut >= 48 && nut <= 55)) {
52.             if (pc->frame_start_found) {
53.                 pc->frame_start_found = 0;
54.                 //返回起始码开始位置
55.                 return i - 5;
56.             }
57.             } else if (nut <= NAL_RASL_R ||
58.                 (nut >= NAL_BLA_W_LP && nut <= NAL_CRA_NUT)) {
59.                 int first_slice_segment_in_pic_flag = buf[i] >> 7;
60.                 if (first_slice_segment_in_pic_flag) {
61.                     if (!pc->frame_start_found) {
62.                         pc->frame_start_found = 1;
63.                     } else { // First slice of next frame found
64.                         pc->frame_start_found = 0;
65.                         //返回起始码开始位置
66.                         return i - 5;
67.                     }
68.                 }
69.             }
70.         }
71.
72.         return END_NOT_FOUND;
73.     }

```

从源代码可以看出, hevc\_find\_frame\_end()使用ParseContext中的state64临时缓存读取的字节。state64是一个uint64\_t类型的变量, 一共可以存储8Byte的数据。函数体的for()循环一次读取一个字节, 读取完成后将该字节放入state64变量中; 接着与起始码“0x000001”进行比较, 如果不相等则继续读取, 如果相等的话则提取NALU Header中nal\_unit\_type信息做相应处理后返回起始码开始的位置。

## parse\_nal\_units()

parse\_nal\_units()用于解析一些NALU (VPS、SPS、PPS) 的信息。该函数的定义位于libavcodec/hevc\_parser.c, 如下所示。

```

1.  /**
2.   * Parse NAL units of found picture and decode some basic information.
3.   *

```

```

3.  *
4.  * @param s parser context.
5.  * @param avctx codec context.
6.  * @param buf buffer with field/frame data.
7.  * @param buf_size size of the buffer.
8.  *
9.  * 解析NALU内容 (不解码)
10. * 注释: 雷霄骅
11. * leixiaohua1020@126.com
12. * http://blog.csdn.net/leixiaohua1020
13. *
14. */
15. static inline int parse_nal_units(AVCodecParserContext *s, AVCodecContext *avctx,
16.                                const uint8_t *buf, int buf_size)
17. {
18.     HEVCContext *h = &((HEVCParserContext *)s->priv_data)->h;
19.     GetBitContext *gb = &h->HEVCLC->gb;
20.     SliceHeader *sh = &h->sh;
21.     const uint8_t *buf_end = buf + buf_size;
22.     int state = -1, i;
23.     HEVCNAL *nal;
24.
25.     /* set some sane default values */
26.     s->pict_type = AV_PICTURE_TYPE_I;
27.     s->key_frame = 0;
28.     s->picture_structure = AV_PICTURE_STRUCTURE_UNKNOWN;
29.
30.     h->avctx = avctx;
31.
32.     if (!buf_size)
33.         return 0;
34.
35.     if (h->nals_allocated < 1) {
36.         HEVCNAL *tmp = av_realloc_array(h->nals, 1, sizeof(*tmp));
37.         if (!tmp)
38.             return AVERROR(ENOMEM);
39.         h->nals = tmp;
40.         memset(h->nals, 0, sizeof(*tmp));
41.         h->nals_allocated = 1;
42.     }
43.
44.     nal = &h->nals[0];
45.
46.     for (;;) {
47.         int src_length, consumed;
48.         buf = avpriv_find_start_code(buf, buf_end, &state);
49.         if (--buf + 2 >= buf_end)
50.             break;
51.         src_length = buf_end - buf;
52.
53.         h->nal_unit_type = (*buf >> 1) & 0x3f;
54.         h->temporal_id = (*buf + 1) & 0x07 - 1;
55.         if (h->nal_unit_type <= NAL_CRA_NUT) {
56.             // Do not walk the whole buffer just to decode slice segment header
57.             if (src_length > 20)
58.                 src_length = 20;
59.         }
60.         //类似于H.264解析器中的ff_h264_decode_nal()
61.         consumed = ff_hevc_extract_rbsp(h, buf, src_length, nal);
62.         if (consumed < 0)
63.             return consumed;
64.
65.         init_get_bits8(gb, nal->data + 2, nal->size);
66.         /*
67.          * 几种NALU之间的关系
68.          *
69.          *          +--SS1
70.          *          |
71.          *      +--PPS1<--+
72.          *          |      |
73.          *      +--SPS1<--+ +--SS2
74.          *          |      |
75.          *      VPS<--+ +--PPS2
76.          *          |
77.          *      +--SPS2
78.          */
79.         //解析不同种类的NALU
80.         switch (h->nal_unit_type) {
81.             case NAL_VPS:
82.                 //解析VPS
83.                 //VPS主要传输视频分级信息,有利于兼容可分级视频编码以及多视点视频编码
84.                 ff_hevc_decode_nal_vps(h);
85.                 break;
86.             case NAL_SPS:
87.                 //解析SPS
88.                 ff_hevc_decode_nal_sps(h);
89.                 break;
90.             case NAL_PPS:
91.                 //解析PPS
92.                 ff_hevc_decode_nal_pps(h);
93.                 break;
94.             case NAL_SET_PROPERTY:

```

```

94.     case NAL_SEI_PREFIX:
95.     case NAL_SEI_SUFFIX:
96.         //解析SEI
97.         ff_hevc_decode_nal_sei(h);
98.         break;
99.     case NAL_TRAIL_N:
100.    case NAL_TRAIL_R:
101.    case NAL_TSA_N:
102.    case NAL_TSA_R:
103.    case NAL_STSA_N:
104.    case NAL_STSA_R:
105.    case NAL_RADL_N:
106.    case NAL_RADL_R:
107.    case NAL_RASL_N:
108.    case NAL_RASL_R:
109.    case NAL_BLA_W_LP:
110.    case NAL_BLA_W_RADL:
111.    case NAL_BLA_N_LP:
112.    case NAL_IDR_W_RADL:
113.    case NAL_IDR_N_LP:
114.    case NAL_CRA_NUT:
115.
116.        //解析 SS Header
117.
118.        //按照解码顺序, 当前SS是否为第1个SS (Slice Segment)
119.        sh->first_slice_in_pic_flag = get_bits1(gb);
120.        s->picture_structure = h->picture_struct;
121.        s->field_order = h->picture_struct;
122.
123.        //IRAP, Intra Random Access Point, 随机介入点
124.        //包括 IDR, CRA, BLA
125.        if (IS_IRAP(h)) {
126.            //设置关键帧
127.            s->key_frame = 1;
128.            sh->no_output_of_prior_pics_flag = get_bits1(gb);
129.        }
130.        //当前Slice引用的PPS的ID号
131.        sh->pps_id = get_ue_golomb(gb);
132.        if (sh->pps_id >= MAX_PPS_COUNT || !h->pps_list[sh->pps_id]) {
133.            av_log(h->avctx, AV_LOG_ERROR, "PPS id out of range: %d\n", sh->pps_id);
134.            return AVERROR_INVALIDDATA;
135.        }
136.        h->pps = (HEVCPPS*)h->pps_list[sh->pps_id]->data;
137.
138.        if (h->pps->sps_id >= MAX_SPS_COUNT || !h->sps_list[h->pps->sps_id]) {
139.            av_log(h->avctx, AV_LOG_ERROR, "SPS id out of range: %d\n", h->pps->sps_id);
140.            return AVERROR_INVALIDDATA;
141.        }
142.        if (h->sps != (HEVCSPS*)h->sps_list[h->pps->sps_id]->data) {
143.            h->sps = (HEVCSPS*)h->sps_list[h->pps->sps_id]->data;
144.            h->vps = (HEVCVPS*)h->vps_list[h->sps->vps_id]->data;
145.        }
146.        //当前Slice不是第一个SS
147.        if (!sh->first_slice_in_pic_flag) {
148.            int slice_address_length;
149.            //当前SS是否依赖SS
150.            if (h->pps->dependent_slice_segments_enabled_flag)
151.                sh->dependent_slice_segment_flag = get_bits1(gb);
152.            else
153.                sh->dependent_slice_segment_flag = 0;
154.
155.            slice_address_length = av_ceil_log2_c(h->sps->ctb_width *
156.                h->sps->ctb_height);
157.            //当前SS中第一个CTU的地址
158.            sh->slice_segment_addr = get_bits(gb, slice_address_length);
159.            if (sh->slice_segment_addr >= h->sps->ctb_width * h->sps->ctb_height) {
160.                av_log(h->avctx, AV_LOG_ERROR, "Invalid slice segment address: %u.\n",
161.                    sh->slice_segment_addr);
162.                return AVERROR_INVALIDDATA;
163.            }
164.        } else
165.            sh->dependent_slice_segment_flag = 0; //独立SS
166.
167.        if (sh->dependent_slice_segment_flag) //依赖SS
168.            break;
169.
170.        for (i = 0; i < h->pps->num_extra_slice_header_bits; i++)
171.            skip_bits(gb, 1); // slice_reserved_undetermined_flag[]
172.
173.        //slice type定义:
174.        // 0: B Slice
175.        // 1: P Slice
176.        // 2: I Slice
177.        sh->slice_type = get_ue_golomb(gb); //
178.        if (!(sh->slice_type == I_SLICE || sh->slice_type == P_SLICE ||
179.            sh->slice_type == B_SLICE)) {
180.            av_log(h->avctx, AV_LOG_ERROR, "Unknown slice type: %d.\n",
181.                sh->slice_type);
182.            return AVERROR_INVALIDDATA;
183.        }
184.        s->pict_type = sh->slice_type == B_SLICE ? AV_PICTURE_TYPE_B :
185.            sh->slice_type == P_SLICE ? AV_PICTURE_TYPE_P :

```

```

186.                                     AV_PICTURE_TYPE_I;
187.
188.         if (h->pps->output_flag_present_flag)
189.             sh->pic_output_flag = get_bits1(gb);
190.
191.         if (h->sps->separate_colour_plane_flag)
192.             sh->colour_plane_id = get_bits(gb, 2);
193.
194.         if (!IS_IDR(h)) {
195.             //不是IDR, 则计算POC
196.             sh->pic_order_cnt_lsb = get_bits(gb, h->sps->log2_max_poc_lsb);
197.             s->output_picture_number = h->poc = ff_hevc_compute_poc(h, sh->pic_order_cnt_lsb);
198.         } else
199.             s->output_picture_number = h->poc = 0;
200.
201.         if (h->temporal_id == 0 &&
202.             h->nal_unit_type != NAL_TRAIL_N &&
203.             h->nal_unit_type != NAL_TSA_N &&
204.             h->nal_unit_type != NAL_STSA_N &&
205.             h->nal_unit_type != NAL_RADL_N &&
206.             h->nal_unit_type != NAL_RASL_N &&
207.             h->nal_unit_type != NAL_RADL_R &&
208.             h->nal_unit_type != NAL_RASL_R)
209.             h->pocTid0 = h->poc;
210.
211.         return 0; /* no need to evaluate the rest */
212.     }
213.     buf += consumed;
214. }
215. /* didn't find a picture! */
216. av_log(h->avctx, AV_LOG_ERROR, "missing picture in access unit\n");
217. return -1;
218. }

```

从源代码可以看出, parse\_nal\_units()根据nal\_unit\_type的不同, 调用不同的解析函数进行处理。例如:

- 解析VPS的时候调用ff\_hevc\_decode\_nal\_vps()
- 解析SPS的时候调用ff\_hevc\_decode\_nal\_sps()
- 解析PPS的时候调用ff\_hevc\_decode\_nal\_pps()
- 解析SEI的时候调用ff\_hevc\_decode\_nal\_sei()
- 解析SS Header的一部分信息。

下文简单分析这几种NALU的解析函数。

## ff\_hevc\_decode\_nal\_vps()

目前还没有研究过VPS, 所以没有分析该函数。

## ff\_hevc\_decode\_nal\_sps()

ff\_hevc\_decode\_nal\_sps()用于解析HEVC码流中的SPS。该函数的定义位于libavcodec\hevc\_ps.c, 如下所示。

```

1.  //解析SPS
2.  int ff_hevc_decode_nal_sps(HEVCContext *s)
3.  {
4.      const AVPixFmtDescriptor *desc;
5.      GetBitContext *gb = &s->HEVCCLc->gb;
6.      int ret = 0;
7.      unsigned int sps_id = 0;
8.      int log2_diff_max_min_transform_block_size;
9.      int bit_depth_chroma, start, vui_present, sublayer_ordering_info;
10.     int i;
11.
12.     HEVCSPS *sps;
13.     AVBufferRef *sps_buf = av_buffer_allocz(sizeof(*sps));
14.
15.     if (!sps_buf)
16.         return AERROR(ENOMEM);
17.     sps = (HEVCSPS*)sps_buf->data;
18.
19.     av_log(s->avctx, AV_LOG_DEBUG, "Decoding SPS\n");
20.
21.     // Coded parameters
22.     // 当前引用的VPS的ID
23.     sps->vps_id = get_bits(gb, 4);
24.     if (sps->vps_id >= MAX_VPS_COUNT) {
25.         av_log(s->avctx, AV_LOG_ERROR, "VPS id out of range: %d\n", sps->vps_id);
26.         ret = AERROR_INVALIDDATA;
27.         goto err;
28.     }
29.
30.     if (!s->vps_list[sps->vps_id]) {
31.         av_log(s->avctx, AV_LOG_ERROR, "VPS %d does not exist\n",

```



```

32.         sps->vps_id);
33.         ret = AERROR_INVALIDDATA;
34.         goto err;
35.     }
36.     //时域子层的最大数目
37.     sps->max_sub_layers = get_bits(gb, 3) + 1;
38.     if (sps->max_sub_layers > MAX_SUB_LAYERS) {
39.         av_log(s->avctx, AV_LOG_ERROR, "sps_max_sub_layers out of range: %d\n",
40.             sps->max_sub_layers);
41.         ret = AERROR_INVALIDDATA;
42.         goto err;
43.     }
44.
45.     skip_bits1(gb); // temporal_id_nesting_flag
46.
47.     if (parse_ptl(s, &sps->ptl, sps->max_sub_layers) < 0)
48.         goto err;
49.     //当前SPS的ID
50.     sps_id = get_ue_golomb_long(gb);
51.     if (sps_id >= MAX_SPS_COUNT) {
52.         av_log(s->avctx, AV_LOG_ERROR, "SPS id out of range: %d\n", sps_id);
53.         ret = AERROR_INVALIDDATA;
54.         goto err;
55.     }
56.     /*
57.      * chroma_format_idc色度取样格式
58.      * 0: Y
59.      * 1: YUV420P
60.      * 2: YUV422P
61.      * 3: YUV444P
62.      */
63.     sps->chroma_format_idc = get_ue_golomb_long(gb);
64.     if (!(sps->chroma_format_idc == 1 || sps->chroma_format_idc == 2 || sps->chroma_format_idc == 3)) {
65.         avpriv_report_missing_feature(s->avctx, "chroma_format_idc != {1, 2, 3}\n");
66.         ret = AERROR_PATCHWELCOME;
67.         goto err;
68.     }
69.     //YUV444的时候, 标记是否对3个分量单独编码
70.     if (sps->chroma_format_idc == 3)
71.         sps->separate_colour_plane_flag = get_bits1(gb);
72.
73.     if (sps->separate_colour_plane_flag)
74.         sps->chroma_format_idc = 0;
75.     //宽和高
76.     sps->width = get_ue_golomb_long(gb);
77.     sps->height = get_ue_golomb_long(gb);
78.     if ((ret = av_image_check_size(sps->width,
79.                                     sps->height, 0, s->avctx)) < 0)
80.         goto err;
81.     //裁剪相关
82.     if (get_bits1(gb)) { // pic_conformance_flag
83.         //TODO: * 2 is only valid for 420
84.         sps->pic_conf_win.left_offset = get_ue_golomb_long(gb) * 2;
85.         sps->pic_conf_win.right_offset = get_ue_golomb_long(gb) * 2;
86.         sps->pic_conf_win.top_offset = get_ue_golomb_long(gb) * 2;
87.         sps->pic_conf_win.bottom_offset = get_ue_golomb_long(gb) * 2;
88.
89.         if (s->avctx->flags2 & CODEC_FLAG2_IGNORE_CROP) {
90.             av_log(s->avctx, AV_LOG_DEBUG,
91.                 "discarding sps conformance window, "
92.                 "original values are l:%u r:%u t:%u b:%u\n",
93.                 sps->pic_conf_win.left_offset,
94.                 sps->pic_conf_win.right_offset,
95.                 sps->pic_conf_win.top_offset,
96.                 sps->pic_conf_win.bottom_offset);
97.
98.             sps->pic_conf_win.left_offset =
99.                 sps->pic_conf_win.right_offset =
100.                 sps->pic_conf_win.top_offset =
101.                 sps->pic_conf_win.bottom_offset = 0;
102.         }
103.         sps->output_window = sps->pic_conf_win;
104.     }
105.     //亮度像素的颜色位深
106.     sps->bit_depth = get_ue_golomb_long(gb) + 8;
107.     //色度像素的颜色位深
108.     bit_depth_chroma = get_ue_golomb_long(gb) + 8;
109.     if (bit_depth_chroma != sps->bit_depth) {
110.         av_log(s->avctx, AV_LOG_ERROR,
111.             "Luma bit depth (%d) is different from chroma bit depth (%d), "
112.             "this is unsupported.\n",
113.             sps->bit_depth, bit_depth_chroma);
114.         ret = AERROR_INVALIDDATA;
115.         goto err;
116.     }
117.     //根据颜色位深和色度采样格式设定pix_fmt
118.     switch (sps->bit_depth) {
119.     case 8:
120.         if (sps->chroma_format_idc == 1) sps->pix_fmt = AV_PIX_FMT_YUV420P;
121.         if (sps->chroma_format_idc == 2) sps->pix_fmt = AV_PIX_FMT_YUV422P;
122.         if (sps->chroma_format_idc == 3) sps->pix_fmt = AV_PIX_FMT_YUV444P;

```

```

123.     break;
124. case 9:
125.     if (sps->chroma_format_idc == 1) sps->pix_fmt = AV_PIX_FMT_YUV420P9;
126.     if (sps->chroma_format_idc == 2) sps->pix_fmt = AV_PIX_FMT_YUV422P9;
127.     if (sps->chroma_format_idc == 3) sps->pix_fmt = AV_PIX_FMT_YUV444P9;
128.     break;
129. case 10:
130.     if (sps->chroma_format_idc == 1) sps->pix_fmt = AV_PIX_FMT_YUV420P10;
131.     if (sps->chroma_format_idc == 2) sps->pix_fmt = AV_PIX_FMT_YUV422P10;
132.     if (sps->chroma_format_idc == 3) sps->pix_fmt = AV_PIX_FMT_YUV444P10;
133.     break;
134. case 12:
135.     if (sps->chroma_format_idc == 1) sps->pix_fmt = AV_PIX_FMT_YUV420P12;
136.     if (sps->chroma_format_idc == 2) sps->pix_fmt = AV_PIX_FMT_YUV422P12;
137.     if (sps->chroma_format_idc == 3) sps->pix_fmt = AV_PIX_FMT_YUV444P12;
138.     break;
139. default:
140.     av_log(s->avctx, AV_LOG_ERROR,
141.            "4:2:0, 4:2:2, 4:4:4 supports are currently specified for 8, 10 and 12 bits.\n");
142.     ret = AERROR_PATCHWELCOME;
143.     goto err;
144. }
145.
146. desc = av_pix_fmt_desc_get(sps->pix_fmt);
147. if (!desc) {
148.     ret = AERROR(EINVAL);
149.     goto err;
150. }
151.
152. sps->hshift[0] = sps->vshift[0] = 0;
153. sps->hshift[2] = sps->hshift[1] = desc->log2_chroma_w;
154. sps->vshift[2] = sps->vshift[1] = desc->log2_chroma_h;
155.
156. sps->pixel_shift = sps->bit_depth > 8;
157.
158. //用于计算POC
159. sps->log2_max_poc_lsb = get_ue_golomb_long(gb) + 4;
160. if (sps->log2_max_poc_lsb > 16) {
161.     av_log(s->avctx, AV_LOG_ERROR, "log2_max_pic_order_cnt_lsb_minus4 out range: %d\n",
162.            sps->log2_max_poc_lsb - 4);
163.     ret = AERROR_INVALIDDATA;
164.     goto err;
165. }
166.
167. sublayer_ordering_info = get_bits1(gb);
168. start = sublayer_ordering_info ? 0 : sps->max_sub_layers - 1;
169. for (i = start; i < sps->max_sub_layers; i++) {
170.     sps->temporal_layer[i].max_dec_pic_buffering = get_ue_golomb_long(gb) + 1;
171.     sps->temporal_layer[i].num_reorder_pics = get_ue_golomb_long(gb);
172.     sps->temporal_layer[i].max_latency_increase = get_ue_golomb_long(gb) - 1;
173.     if (sps->temporal_layer[i].max_dec_pic_buffering > MAX_DPB_SIZE) {
174.         av_log(s->avctx, AV_LOG_ERROR, "sps_max_dec_pic_buffering_minus1 out of range: %d\n",
175.                sps->temporal_layer[i].max_dec_pic_buffering - 1);
176.         ret = AERROR_INVALIDDATA;
177.         goto err;
178.     }
179.     if (sps->temporal_layer[i].num_reorder_pics > sps->temporal_layer[i].max_dec_pic_buffering - 1) {
180.         av_log(s->avctx, AV_LOG_WARNING, "sps_max_num_reorder_pics out of range: %d\n",
181.                sps->temporal_layer[i].num_reorder_pics);
182.         if (s->avctx->err_recognition & AV_EF_EXPLODE ||
183.             sps->temporal_layer[i].num_reorder_pics > MAX_DPB_SIZE - 1) {
184.             ret = AERROR_INVALIDDATA;
185.             goto err;
186.         }
187.         sps->temporal_layer[i].max_dec_pic_buffering = sps->temporal_layer[i].num_reorder_pics + 1;
188.     }
189. }
190.
191. if (!sublayer_ordering_info) {
192.     for (i = 0; i < start; i++) {
193.         sps->temporal_layer[i].max_dec_pic_buffering = sps->temporal_layer[start].max_dec_pic_buffering;
194.         sps->temporal_layer[i].num_reorder_pics = sps->temporal_layer[start].num_reorder_pics;
195.         sps->temporal_layer[i].max_latency_increase = sps->temporal_layer[start].max_latency_increase;
196.     }
197. }
198. //亮度编码块-最小尺寸
199. sps->log2_min_cb_size = get_ue_golomb_long(gb) + 3;
200. //亮度编码块-最大尺寸和最小尺寸插值
201. sps->log2_diff_max_min_coding_block_size = get_ue_golomb_long(gb);
202. //亮度变换块-最小尺寸
203. sps->log2_min_tb_size = get_ue_golomb_long(gb) + 2;
204. //亮度变换块-最大尺寸和最小尺寸插值
205. log2_diff_max_min_transform_block_size = get_ue_golomb_long(gb);
206. sps->log2_max_trafo_size = log2_diff_max_min_transform_block_size +
207.                             sps->log2_min_tb_size;
208.
209. if (sps->log2_min_tb_size >= sps->log2_min_cb_size) {
210.     av_log(s->avctx, AV_LOG_ERROR, "Invalid value for log2_min_tb_size");
211.     ret = AERROR_INVALIDDATA;
212.     goto err;
213. }

```

```

214. //帧间预测变换块-最大划分深度
215. sps->max_transform_hierarchy_depth_inter = get_ue_golomb_long(gb);
216. //帧内预测变换块-最大划分深度
217. sps->max_transform_hierarchy_depth_intra = get_ue_golomb_long(gb);
218. //是否使用量化矩阵
219. sps->scaling_list_enable_flag = get_bits1(gb);
220. if (sps->scaling_list_enable_flag) {
221.     set_default_scaling_list_data(&sps->scaling_list);
222.
223.     if (get_bits1(gb)) {
224.         ret = scaling_list_data(s, &sps->scaling_list, sps);
225.         if (ret < 0)
226.             goto err;
227.     }
228. }
229. //是否使用非对称划分模式
230. sps->amp_enabled_flag = get_bits1(gb);
231. //是否在去块效应滤波过程中使用样点自适应补偿SAO
232. sps->sao_enabled = get_bits1(gb);
233. //允许PCM编码
234. sps->pcm_enabled_flag = get_bits1(gb);
235. //有关PCM编码的参数
236. if (sps->pcm_enabled_flag) {
237.     sps->pcm.bit_depth = get_bits(gb, 4) + 1;
238.     sps->pcm.bit_depth_chroma = get_bits(gb, 4) + 1;
239.     sps->pcm.log2_min_pcm_cb_size = get_ue_golomb_long(gb) + 3;
240.     sps->pcm.log2_max_pcm_cb_size = sps->pcm.log2_min_pcm_cb_size +
241.                                     get_ue_golomb_long(gb);
242.     if (sps->pcm.bit_depth > sps->bit_depth) {
243.         av_log(s->avctx, AV_LOG_ERROR,
244.              "PCM bit depth (%d) is greater than normal bit depth (%d)\n",
245.              sps->pcm.bit_depth, sps->bit_depth);
246.         ret = AERROR_INVALIDDATA;
247.         goto err;
248.     }
249.
250.     sps->pcm.loop_filter_disable_flag = get_bits1(gb);
251. }
252. //短期参考num_short_term_ref_pic_set
253. sps->nb_st_rps = get_ue_golomb_long(gb);
254. if (sps->nb_st_rps > MAX_SHORT_TERM_RPS_COUNT) {
255.     av_log(s->avctx, AV_LOG_ERROR, "Too many short term RPS: %d.\n",
256.          sps->nb_st_rps);
257.     ret = AERROR_INVALIDDATA;
258.     goto err;
259. }
260. for (i = 0; i < sps->nb_st_rps; i++) {
261.     if ((ret = ff_hevc_decode_short_term_rps(s, &sps->st_rps[i],
262.          sps, 0)) < 0)
263.         goto err;
264. }
265. //长期参考num_long_term_ref_pic_set
266. sps->long_term_ref_pics_present_flag = get_bits1(gb);
267. if (sps->long_term_ref_pics_present_flag) {
268.     sps->num_long_term_ref_pics_sps = get_ue_golomb_long(gb);
269.     if (sps->num_long_term_ref_pics_sps > 31U) {
270.         av_log(0, AV_LOG_ERROR, "num_long_term_ref_pics_sps %d is out of range.\n",
271.              sps->num_long_term_ref_pics_sps);
272.         goto err;
273.     }
274.     for (i = 0; i < sps->num_long_term_ref_pics_sps; i++) {
275.         sps->lt_ref_pic_poc_lsb_sps[i] = get_bits(gb, sps->log2_max_poc_lsb);
276.         sps->used_by_curr_pic_lt_sps_flag[i] = get_bits1(gb);
277.     }
278. }
279. //是否使用时域MV预测
280. sps->sps_temporal_mvp_enabled_flag = get_bits1(gb);
281. //滤波过程是否使用双线性插值
282. sps->sps_strong_intra_smoothing_enable_flag = get_bits1(gb);
283. sps->vui.sar = (AVRational){0, 1};
284. vui_present = get_bits1(gb);
285. if (vui_present)
286.     decode_vui(s, sps);
287.
288. if (get_bits1(gb)) { // sps_extension_flag
289.     int sps_extension_flag[1];
290.     for (i = 0; i < 1; i++)
291.         sps_extension_flag[i] = get_bits1(gb);
292.     skip_bits(gb, 7); //sps_extension_7bits = get_bits(gb, 7);
293.     if (sps_extension_flag[0]) {
294.         int extended_precision_processing_flag;
295.         int high_precision_offsets_enabled_flag;
296.         int cabac_bypass_alignment_enabled_flag;
297.
298.         sps->transform_skip_rotation_enabled_flag = get_bits1(gb);
299.         sps->transform_skip_context_enabled_flag = get_bits1(gb);
300.         sps->implicit_rdpem_enabled_flag = get_bits1(gb);
301.
302.         sps->explicit_rdpem_enabled_flag = get_bits1(gb);
303.
304.         extended_precision_processing_flag = get_bits1(gb);
305.         if (extended_precision_processing_flag)

```

```

305.         1T (extended_precision_processing_flag)
306.         av_log(s->avctx, AV_LOG_WARNING,
307.             "extended_precision_processing_flag not yet implemented\n");
308.
309.         sps->intra_smoothing_disabled_flag = get_bits1(gb);
310.         high_precision_offsets_enabled_flag = get_bits1(gb);
311.         if (high_precision_offsets_enabled_flag)
312.             av_log(s->avctx, AV_LOG_WARNING,
313.                 "high_precision_offsets_enabled_flag not yet implemented\n");
314.
315.         sps->persistent_rice_adaptation_enabled_flag = get_bits1(gb);
316.
317.         cabac_bypass_alignment_enabled_flag = get_bits1(gb);
318.         if (cabac_bypass_alignment_enabled_flag)
319.             av_log(s->avctx, AV_LOG_WARNING,
320.                 "cabac_bypass_alignment_enabled_flag not yet implemented\n");
321.     }
322. }
323. if (s->apply_defdispwin) {
324.     sps->output_window.left_offset += sps->vui.def_disp_win.left_offset;
325.     sps->output_window.right_offset += sps->vui.def_disp_win.right_offset;
326.     sps->output_window.top_offset += sps->vui.def_disp_win.top_offset;
327.     sps->output_window.bottom_offset += sps->vui.def_disp_win.bottom_offset;
328. }
329. if (sps->output_window.left_offset & (0x1F >> (sps->pixel_shift)) &&
330.     !(s->avctx->flags & CODEC_FLAG_UNALIGNED)) {
331.     sps->output_window.left_offset &= ~(0x1F >> (sps->pixel_shift));
332.     av_log(s->avctx, AV_LOG_WARNING, "Reducing left output window to %d "
333.         "chroma samples to preserve alignment.\n",
334.         sps->output_window.left_offset);
335. }
336. sps->output_width = sps->width -
337.     (sps->output_window.left_offset + sps->output_window.right_offset);
338. sps->output_height = sps->height -
339.     (sps->output_window.top_offset + sps->output_window.bottom_offset);
340. if (sps->output_width <= 0 || sps->output_height <= 0) {
341.     av_log(s->avctx, AV_LOG_WARNING, "Invalid visible frame dimensions: %dx%d.\n",
342.         sps->output_width, sps->output_height);
343.     if (s->avctx->err_recognition & AV_EF_EXPLODE) {
344.         ret = AVERROUR_INVALIDDATA;
345.         goto err;
346.     }
347.     av_log(s->avctx, AV_LOG_WARNING,
348.         "Displaying the whole video surface.\n");
349.     memset(&sps->pic_conf_win, 0, sizeof(sps->pic_conf_win));
350.     memset(&sps->output_window, 0, sizeof(sps->output_window));
351.     sps->output_width = sps->width;
352.     sps->output_height = sps->height;
353. }
354.
355. // Inferred parameters
356. // 推算出来的参数
357. sps->log2_ctb_size = sps->log2_min_cb_size +
358.     sps->log2_diff_max_min_coding_block_size;
359. sps->log2_min_pu_size = sps->log2_min_cb_size - 1;
360.
361. sps->ctb_width = (sps->width + (1 << sps->log2_ctb_size) - 1) >> sps->log2_ctb_size;
362. sps->ctb_height = (sps->height + (1 << sps->log2_ctb_size) - 1) >> sps->log2_ctb_size;
363. sps->ctb_size = sps->ctb_width * sps->ctb_height;
364.
365. sps->min_cb_width = sps->width >> sps->log2_min_cb_size;
366. sps->min_cb_height = sps->height >> sps->log2_min_cb_size;
367. sps->min_tb_width = sps->width >> sps->log2_min_tb_size;
368. sps->min_tb_height = sps->height >> sps->log2_min_tb_size;
369. sps->min_pu_width = sps->width >> sps->log2_min_pu_size;
370. sps->min_pu_height = sps->height >> sps->log2_min_pu_size;
371. sps->tb_mask = (1 << (sps->log2_ctb_size - sps->log2_min_tb_size)) - 1;
372.
373. sps->qp_bd_offset = 6 * (sps->bit_depth - 8);
374.
375. if (sps->width & ((1 << sps->log2_min_cb_size) - 1) ||
376.     sps->height & ((1 << sps->log2_min_cb_size) - 1)) {
377.     av_log(s->avctx, AV_LOG_ERROR, "Invalid coded frame dimensions.\n");
378.     goto err;
379. }
380.
381. if (sps->log2_ctb_size > MAX_LOG2_CTB_SIZE) {
382.     av_log(s->avctx, AV_LOG_ERROR, "CTB size out of range: 2^%d\n", sps->log2_ctb_size);
383.     goto err;
384. }
385. if (sps->max_transform_hierarchy_depth_inter > sps->log2_ctb_size - sps->log2_min_tb_size) {
386.     av_log(s->avctx, AV_LOG_ERROR, "max_transform_hierarchy_depth_inter out of range: %d\n",
387.         sps->max_transform_hierarchy_depth_inter);
388.     goto err;
389. }
390. if (sps->max_transform_hierarchy_depth_intra > sps->log2_ctb_size - sps->log2_min_tb_size) {
391.     av_log(s->avctx, AV_LOG_ERROR, "max_transform_hierarchy_depth_intra out of range: %d\n",
392.         sps->max_transform_hierarchy_depth_intra);
393.     goto err;
394. }
395. if (sps->log2_max_trafo_size > FFMIN(sps->log2_ctb_size, 5)) {
396.     av_log(s->avctx, AV_LOG_ERROR,

```

```

397.         av_log(s->avctx, AV_LOG_ERROR,
398.             "max transform block size out of range: %d\n",
399.             sps->log2_max_trafo_size);
400.         goto err;
401.     }
402.     if (get_bits_left(gb) < 0) {
403.         av_log(s->avctx, AV_LOG_ERROR,
404.             "Overread SPS by %d bits\n", -get_bits_left(gb));
405.         goto err;
406.     }
407.
408.     if (s->avctx->debug & FF_DEBUG_BITSTREAM) {
409.         av_log(s->avctx, AV_LOG_DEBUG,
410.             "Parsed SPS: id %d; coded wxh: %dx%d; "
411.             "cropped wxh: %dx%d; pix_fmt: %s.\n",
412.             sps_id, sps->width, sps->height,
413.             sps->output_width, sps->output_height,
414.             av_get_pix_fmt_name(sps->pix_fmt));
415.     }
416.
417.     /* check if this is a repeat of an already parsed SPS, then keep the
418.      * original one.
419.      * otherwise drop all PPSES that depend on it */
420.     if (s->sps_list[sps_id] &&
421.         !memcmp(s->sps_list[sps_id]->data, sps_buf->data, sps_buf->size)) {
422.         av_buffer_unref(&sps_buf);
423.     } else {
424.         for (i = 0; i < FF_ARRAY_ELEMS(s->pps_list); i++) {
425.             if (s->pps_list[i] && ((HEVCPPS*)s->pps_list[i]->data)->sps_id == sps_id)
426.                 av_buffer_unref(&s->pps_list[i]);
427.         }
428.         if (s->sps_list[sps_id] && s->sps == (HEVCSPS*)s->sps_list[sps_id]->data) {
429.             av_buffer_unref(&s->current_sps);
430.             s->current_sps = av_buffer_ref(s->sps_list[sps_id]);
431.             if (!s->current_sps)
432.                 s->sps = NULL;
433.         }
434.         av_buffer_unref(&s->sps_list[sps_id]);
435.         s->sps_list[sps_id] = sps_buf;
436.     }
437.
438.     return 0;
439.
440. err:
441.     av_buffer_unref(&sps_buf);
442.     return ret;
443. }

```

解析SPS源代码并不是很有“技术含量”。只要参考ITU-T的《HEVC标准》就可以理解了，不再做过多详细的分析。

## ff\_hevc\_decode\_nal\_pps()

ff\_hevc\_decode\_nal\_pps()用于解析HEVC码流中的PPS。该函数的定义位于libavcodec/hevc\_ps.c，如下所示。

```

1. //解析PPS
2. int ff_hevc_decode_nal_pps(HEVCContext *s)
3. {
4.     GetBitContext *gb = &s->HEVClc->gb;
5.     HEVCSPS *sps = NULL;
6.     int pic_area_in_ctbs;
7.     int log2_diff_ctb_min_tb_size;
8.     int i, j, x, y, ctb_addr_rs, tile_id;
9.     int ret = 0;
10.    unsigned int pps_id = 0;
11.
12.    AVBufferRef *pps_buf;
13.    HEVCPPS *pps = av_mallocz(sizeof(*pps));
14.
15.    if (!pps)
16.        return AVERROR(ENOMEM);
17.
18.    pps_buf = av_buffer_create((uint8_t *)pps, sizeof(*pps),
19.                               hevc_pps_free, NULL, 0);
20.    if (!pps_buf) {
21.        av_freep(&pps);
22.        return AVERROR(ENOMEM);
23.    }
24.
25.    av_log(s->avctx, AV_LOG_DEBUG, "Decoding PPS\n");
26.
27.    // Default values
28.    // 默认值
29.    pps->loop_filter_across_tiles_enabled_flag = 1;
30.    pps->num_tile_columns = 1;
31.    pps->num_tile_rows = 1;
32.    pps->uniform_spacing_flag = 1;

```

```

33.     pps->disable_dbf = 0;
34.     pps->beta_offset = 0;
35.     pps->tc_offset = 0;
36.     pps->log2_max_transform_skip_block_size = 2;
37.
38.     // Coded parameters
39.     //当前PPS的ID
40.     pps_id = get_ue_golomb_long(gb);
41.     if (pps_id >= MAX_PPS_COUNT) {
42.         av_log(s->avctx, AV_LOG_ERROR, "PPS id out of range: %d\n", pps_id);
43.         ret = AERROR_INVALIDDATA;
44.         goto err;
45.     }
46.     //引用的SPS的ID
47.     pps->sps_id = get_ue_golomb_long(gb);
48.     if (pps->sps_id >= MAX_SPS_COUNT) {
49.         av_log(s->avctx, AV_LOG_ERROR, "SPS id out of range: %d\n", pps->sps_id);
50.         ret = AERROR_INVALIDDATA;
51.         goto err;
52.     }
53.     if (!s->sps_list[pps->sps_id]) {
54.         av_log(s->avctx, AV_LOG_ERROR, "SPS %u does not exist.\n", pps->sps_id);
55.         ret = AERROR_INVALIDDATA;
56.         goto err;
57.     }
58.     sps = (HEVCSPS *)s->sps_list[pps->sps_id]->data;
59.     //判断当前Slice是否包含依赖片
60.     pps->dependent_slice_segments_enabled_flag = get_bits1(gb);
61.     pps->output_flag_present_flag = get_bits1(gb);
62.     pps->num_extra_slice_header_bits = get_bits(gb, 3);
63.
64.     pps->sign_data_hiding_flag = get_bits1(gb);
65.     //在CABAC中用何种方式确定上下文变量的初始值
66.     pps->cabac_init_present_flag = get_bits1(gb);
67.     //list0中参考图像数目的默认最大值
68.     pps->num_ref_idx_l0_default_active = get_ue_golomb_long(gb) + 1;
69.     //list1中参考图像数目的默认最大值
70.     pps->num_ref_idx_l1_default_active = get_ue_golomb_long(gb) + 1;
71.     //亮度分量QP的初始值
72.     pps->pic_init_qp_minus26 = get_se_golomb(gb);
73.
74.     pps->constrained_intra_pred_flag = get_bits1(gb);
75.     pps->transform_skip_enabled_flag = get_bits1(gb);
76.
77.     pps->cu_qp_delta_enabled_flag = get_bits1(gb);
78.     pps->diff_cu_qp_delta_depth = 0;
79.     if (pps->cu_qp_delta_enabled_flag)
80.         pps->diff_cu_qp_delta_depth = get_ue_golomb_long(gb);
81.
82.     if (pps->diff_cu_qp_delta_depth < 0 ||
83.         pps->diff_cu_qp_delta_depth > sps->log2_diff_max_min_coding_block_size) {
84.         av_log(s->avctx, AV_LOG_ERROR, "diff_cu_qp_delta_depth %d is invalid\n",
85.             pps->diff_cu_qp_delta_depth);
86.         ret = AERROR_INVALIDDATA;
87.         goto err;
88.     }
89.
90.     pps->cb_qp_offset = get_se_golomb(gb);
91.     if (pps->cb_qp_offset < -12 || pps->cb_qp_offset > 12) {
92.         av_log(s->avctx, AV_LOG_ERROR, "pps_cb_qp_offset out of range: %d\n",
93.             pps->cb_qp_offset);
94.         ret = AERROR_INVALIDDATA;
95.         goto err;
96.     }
97.     pps->cr_qp_offset = get_se_golomb(gb);
98.     if (pps->cr_qp_offset < -12 || pps->cr_qp_offset > 12) {
99.         av_log(s->avctx, AV_LOG_ERROR, "pps_cr_qp_offset out of range: %d\n",
100.             pps->cr_qp_offset);
101.         ret = AERROR_INVALIDDATA;
102.         goto err;
103.     }
104.     pps->pic_slice_level_chroma_qp_offsets_present_flag = get_bits1(gb);
105.     //P Slice是否使用加权预测
106.     pps->weighted_pred_flag = get_bits1(gb);
107.     //B Slice是否使用加权预测
108.     pps->weighted_bipred_flag = get_bits1(gb);
109.
110.     pps->transquant_bypass_enable_flag = get_bits1(gb);
111.     //是否使用tile
112.     pps->tiles_enabled_flag = get_bits1(gb);
113.     pps->entropy_coding_sync_enabled_flag = get_bits1(gb);
114.
115.     if (pps->tiles_enabled_flag) {
116.         //Tile的列数
117.         pps->num_tile_columns = get_ue_golomb_long(gb) + 1;
118.         //Tile的行数
119.         pps->num_tile_rows = get_ue_golomb_long(gb) + 1;
120.         if (pps->num_tile_columns == 0 ||
121.             pps->num_tile_columns >= sps->width) {
122.             av_log(s->avctx, AV_LOG_ERROR, "num_tile_columns_minus1 out of range: %d\n",
123.                 pps->num_tile_columns - 1);

```

```

124.         ret = AERROR_INVALIDDATA;
125.         goto err;
126.     }
127.     if (pps->num_tile_rows == 0 ||
128.         pps->num_tile_rows >= sps->height) {
129.         av_log(s->avctx, AV_LOG_ERROR, "num_tile_rows minus1 out of range: %d\n",
130.             pps->num_tile_rows - 1);
131.         ret = AERROR_INVALIDDATA;
132.         goto err;
133.     }
134.
135.     pps->column_width = av_malloc_array(pps->num_tile_columns, sizeof(*pps->column_width));
136.     pps->row_height = av_malloc_array(pps->num_tile_rows, sizeof(*pps->row_height));
137.     if (!pps->column_width || !pps->row_height) {
138.         ret = AERROR(ENOMEM);
139.         goto err;
140.     }
141.
142.     pps->uniform_spacing_flag = get_bits1(gb);
143.     if (!pps->uniform_spacing_flag) {
144.         uint64_t sum = 0;
145.         //每个Tile的宽度和高度
146.         for (i = 0; i < pps->num_tile_columns - 1; i++) {
147.             pps->column_width[i] = get_ue_golomb_long(gb) + 1;
148.             sum += pps->column_width[i];
149.         }
150.         if (sum >= sps->ctb_width) {
151.             av_log(s->avctx, AV_LOG_ERROR, "Invalid tile widths.\n");
152.             ret = AERROR_INVALIDDATA;
153.             goto err;
154.         }
155.         pps->column_width[pps->num_tile_columns - 1] = sps->ctb_width - sum;
156.
157.         sum = 0;
158.         for (i = 0; i < pps->num_tile_rows - 1; i++) {
159.             pps->row_height[i] = get_ue_golomb_long(gb) + 1;
160.             sum += pps->row_height[i];
161.         }
162.         if (sum >= sps->ctb_height) {
163.             av_log(s->avctx, AV_LOG_ERROR, "Invalid tile heights.\n");
164.             ret = AERROR_INVALIDDATA;
165.             goto err;
166.         }
167.         pps->row_height[pps->num_tile_rows - 1] = sps->ctb_height - sum;
168.     }
169.     pps->loop_filter_across_tiles_enabled_flag = get_bits1(gb);
170. }
171.
172. pps->seq_loop_filter_across_slices_enabled_flag = get_bits1(gb);
173. //是否存在去方块滤波的控制信息
174. pps->deblocking_filter_control_present_flag = get_bits1(gb);
175. if (pps->deblocking_filter_control_present_flag) {
176.     pps->deblocking_filter_override_enabled_flag = get_bits1(gb);
177.     //是否使用去方块滤波
178.     pps->disable_dbf = get_bits1(gb);
179.     if (!pps->disable_dbf) {
180.         pps->beta_offset = get_se_golomb(gb) * 2;
181.         pps->tc_offset = get_se_golomb(gb) * 2;
182.         if (pps->beta_offset/2 < -6 || pps->beta_offset/2 > 6) {
183.             av_log(s->avctx, AV_LOG_ERROR, "pps_beta_offset_div2 out of range: %d\n",
184.                 pps->beta_offset/2);
185.             ret = AERROR_INVALIDDATA;
186.             goto err;
187.         }
188.         if (pps->tc_offset/2 < -6 || pps->tc_offset/2 > 6) {
189.             av_log(s->avctx, AV_LOG_ERROR, "pps_tc_offset_div2 out of range: %d\n",
190.                 pps->tc_offset/2);
191.             ret = AERROR_INVALIDDATA;
192.             goto err;
193.         }
194.     }
195. }
196.
197. pps->scaling_list_data_present_flag = get_bits1(gb);
198. if (pps->scaling_list_data_present_flag) {
199.     set_default_scaling_list_data(&pps->scaling_list);
200.     ret = scaling_list_data(s, &pps->scaling_list, sps);
201.     if (ret < 0)
202.         goto err;
203. }
204. pps->lists_modification_present_flag = get_bits1(gb);
205. pps->log2_parallel_merge_level = get_ue_golomb_long(gb) + 2;
206. if (pps->log2_parallel_merge_level > sps->log2_ctb_size) {
207.     av_log(s->avctx, AV_LOG_ERROR, "log2_parallel_merge_level_minus2 out of range: %d\n",
208.         pps->log2_parallel_merge_level - 2);
209.     ret = AERROR_INVALIDDATA;
210.     goto err;
211. }
212.
213. pps->slice_header_extension_present_flag = get_bits1(gb);
214.
215. if (pps->slice_header_extension_present_flag) {

```

```

215.     1t (get_bits1(gb)) { // pps_extension_present_flag
216.         int pps_range_extensions_flag = get_bits1(gb);
217.         /* int pps_extension_7bits = */ get_bits(gb, 7);
218.         if (sps->ptl.general_ptl.profile_idc == FF_PROFILE_HEVC_REXT && pps_range_extensions_flag) {
219.             if ((ret = pps_range_extensions(s, pps, sps)) < 0)
220.                 goto err;
221.         }
222.     }
223.
224.     // Inferred parameters
225.     pps->col_bd = av_malloc_array(pps->num_tile_columns + 1, sizeof(*pps->col_bd));
226.     pps->row_bd = av_malloc_array(pps->num_tile_rows + 1, sizeof(*pps->row_bd));
227.     pps->col_idxX = av_malloc_array(sps->ctb_width, sizeof(*pps->col_idxX));
228.     if (!pps->col_bd || !pps->row_bd || !pps->col_idxX) {
229.         ret = AERROR(ENOMEM);
230.         goto err;
231.     }
232.
233.     if (pps->uniform_spacing_flag) {
234.         if (!pps->column_width) {
235.             pps->column_width = av_malloc_array(pps->num_tile_columns, sizeof(*pps->column_width));
236.             pps->row_height = av_malloc_array(pps->num_tile_rows, sizeof(*pps->row_height));
237.         }
238.         if (!pps->column_width || !pps->row_height) {
239.             ret = AERROR(ENOMEM);
240.             goto err;
241.         }
242.
243.         for (i = 0; i < pps->num_tile_columns; i++) {
244.             pps->column_width[i] = ((i + 1) * sps->ctb_width) / pps->num_tile_columns -
245.                                     (i * sps->ctb_width) / pps->num_tile_columns;
246.         }
247.
248.         for (i = 0; i < pps->num_tile_rows; i++) {
249.             pps->row_height[i] = ((i + 1) * sps->ctb_height) / pps->num_tile_rows -
250.                                   (i * sps->ctb_height) / pps->num_tile_rows;
251.         }
252.     }
253.
254.     pps->col_bd[0] = 0;
255.     for (i = 0; i < pps->num_tile_columns; i++)
256.         pps->col_bd[i + 1] = pps->col_bd[i] + pps->column_width[i];
257.
258.     pps->row_bd[0] = 0;
259.     for (i = 0; i < pps->num_tile_rows; i++)
260.         pps->row_bd[i + 1] = pps->row_bd[i] + pps->row_height[i];
261.
262.     for (i = 0, j = 0; i < sps->ctb_width; i++) {
263.         if (i > pps->col_bd[j])
264.             j++;
265.         pps->col_idxX[i] = j;
266.     }
267.
268.     /**
269.      * 6.5
270.      */
271.     pic_area_in_ctbs = sps->ctb_width * sps->ctb_height;
272.
273.     pps->ctb_addr_rs_to_ts = av_malloc_array(pic_area_in_ctbs, sizeof(*pps->ctb_addr_rs_to_ts));
274.     pps->ctb_addr_ts_to_rs = av_malloc_array(pic_area_in_ctbs, sizeof(*pps->ctb_addr_ts_to_rs));
275.     pps->tile_id = av_malloc_array(pic_area_in_ctbs, sizeof(*pps->tile_id));
276.     pps->min_tb_addr_zs_tab = av_malloc_array((sps->tb_mask+2) * (sps->tb_mask+2), sizeof(*pps->min_tb_addr_zs_tab));
277.     if (!pps->ctb_addr_rs_to_ts || !pps->ctb_addr_ts_to_rs ||
278.         !pps->tile_id || !pps->min_tb_addr_zs_tab) {
279.         ret = AERROR(ENOMEM);
280.         goto err;
281.     }
282.
283.     for (ctb_addr_rs = 0; ctb_addr_rs < pic_area_in_ctbs; ctb_addr_rs++) {
284.         int tb_x = ctb_addr_rs % sps->ctb_width;
285.         int tb_y = ctb_addr_rs / sps->ctb_width;
286.         int tile_x = 0;
287.         int tile_y = 0;
288.         int val = 0;
289.
290.         for (i = 0; i < pps->num_tile_columns; i++) {
291.             if (tb_x < pps->col_bd[i + 1]) {
292.                 tile_x = i;
293.                 break;
294.             }
295.         }
296.
297.         for (i = 0; i < pps->num_tile_rows; i++) {
298.             if (tb_y < pps->row_bd[i + 1]) {
299.                 tile_y = i;
300.                 break;
301.             }
302.         }
303.
304.         for (i = 0; i < tile_x; i++)
305.             val += pps->row_height[tile_y] * pps->column_width[i];
306.         for (i = 0; i < tile_y; i++)

```



```

300.         val += sps->ctb_width * pps->row_height[i];
301.
302.         val += (tb_y - pps->row_bd[tile_y]) * pps->column_width[tile_x] +
303.             tb_x - pps->col_bd[tile_x];
304.
305.         pps->ctb_addr_rs_to_ts[ctb_addr_rs] = val;
306.         pps->ctb_addr_ts_to_rs[val] = ctb_addr_rs;
307.     }
308.
309.     for (j = 0, tile_id = 0; j < pps->num_tile_rows; j++)
310.         for (i = 0; i < pps->num_tile_columns; i++, tile_id++)
311.             for (y = pps->row_bd[j]; y < pps->row_bd[j + 1]; y++)
312.                 for (x = pps->col_bd[i]; x < pps->col_bd[i + 1]; x++)
313.                     pps->tile_id[pps->ctb_addr_rs_to_ts[y * sps->ctb_width + x]] = tile_id;
314.
315.     pps->tile_pos_rs = av_malloc_array(tile_id, sizeof(*pps->tile_pos_rs));
316.     if (!pps->tile_pos_rs) {
317.         ret = AERROR(ENOMEM);
318.         goto err;
319.     }
320.
321.     for (j = 0; j < pps->num_tile_rows; j++)
322.         for (i = 0; i < pps->num_tile_columns; i++)
323.             pps->tile_pos_rs[j * pps->num_tile_columns + i] = pps->row_bd[j] * sps->ctb_width + pps->col_bd[i];
324.
325.     log2_diff_ctb_min_tb_size = sps->log2_ctb_size - sps->log2_min_tb_size;
326.     pps->min_tb_addr_zs = &pps->min_tb_addr_zs_tab[1*(sps->tb_mask+2)+1];
327.     for (y = 0; y < sps->tb_mask+2; y++) {
328.         pps->min_tb_addr_zs_tab[y*(sps->tb_mask+2)] = -1;
329.         pps->min_tb_addr_zs_tab[y] = -1;
330.     }
331.     for (y = 0; y < sps->tb_mask+1; y++) {
332.         for (x = 0; x < sps->tb_mask+1; x++) {
333.             int tb_x = x >> log2_diff_ctb_min_tb_size;
334.             int tb_y = y >> log2_diff_ctb_min_tb_size;
335.             int ctb_addr_rs = sps->ctb_width * tb_y + tb_x;
336.             int val = pps->ctb_addr_rs_to_ts[ctb_addr_rs] <<
337.                 (log2_diff_ctb_min_tb_size * 2);
338.             for (i = 0; i < log2_diff_ctb_min_tb_size; i++) {
339.                 int m = 1 << i;
340.                 val += (m & x ? m * m : 0) + (m & y ? 2 * m * m : 0);
341.             }
342.             pps->min_tb_addr_zs[y * (sps->tb_mask+2) + x] = val;
343.         }
344.     }
345.
346.     if (get_bits_left(gb) < 0) {
347.         av_log(s->avctx, AV_LOG_ERROR,
348.             "Overread PPS by %d bits\n", -get_bits_left(gb));
349.         goto err;
350.     }
351.
352.     av_buffer_unref(&s->pps_list[pps_id]);
353.     s->pps_list[pps_id] = pps_buf;
354.
355.     return 0;
356.
357. err:
358.     av_buffer_unref(&pps_buf);
359.     return ret;
360. }

```

与解析SPS类似，解析PPS源代码并不是很有“技术含量”。只要参考ITU-T的《H.264标准》就可以理解了，不再做过多详细的分析。

## ff\_hevc\_decode\_nal\_sei()

ff\_hevc\_decode\_nal\_sei()用于解析HEVC码流中的SEI。该函数的定义位于libavcodec\hevc\_sei.c，如下所示。

```

1. //解析SEI
2. int ff_hevc_decode_nal_sei(HEVCContext *s)
3. {
4.     int ret;
5.
6.     do {
7.         //解析SEI信息
8.         ret = decode_nal_sei_message(s);
9.         if (ret < 0)
10.             return(AERROR(ENOMEM));
11.     } while (more_rbsp_data(&s->HEVCLc->gb));
12.     return 1;
13. }

```

从源代码可以看出，ff\_hevc\_decode\_nal\_sei()在一个do while循环中调用了另外一个函数decode\_nal\_sei\_message()解析SEI信息。

## decode\_nal\_sei\_message()

decode\_nal\_sei\_message()用于解析SEI信息，它的定义如下。

```
[cpp]
1. //解析SEI信息
2. static int decode_nal_sei_message(HEVCContext *s)
3. {
4.     GetBitContext *gb = &s->HEVCLc->gb;
5.
6.     int payload_type = 0;
7.     int payload_size = 0;
8.     int byte = 0xFF;
9.     av_log(s->avctx, AV_LOG_DEBUG, "Decoding SEI\n");
10.
11.     while (byte == 0xFF) {
12.         byte = get_bits(gb, 8);
13.         payload_type += byte;
14.     }
15.     byte = 0xFF;
16.     while (byte == 0xFF) {
17.         byte = get_bits(gb, 8);
18.         payload_size += byte;
19.     }
20.     if (s->nal_unit_type == NAL_SEI_PREFIX) {
21.         if (payload_type == 256 /*&& s->decode_checksum_sei*/) {
22.             decode_nal_sei_decoded_picture_hash(s);
23.         } else if (payload_type == 45) {
24.             decode_nal_sei_frame_packing_arrangement(s);
25.         } else if (payload_type == 47) {
26.             decode_nal_sei_display_orientation(s);
27.         } else if (payload_type == 1){
28.             int ret = decode_pic_timing(s);
29.             av_log(s->avctx, AV_LOG_DEBUG, "Skipped PREFIX SEI %d\n", payload_type);
30.             skip_bits(gb, 8 * payload_size);
31.             return ret;
32.         } else if (payload_type == 129){
33.             active_parameter_sets(s);
34.             av_log(s->avctx, AV_LOG_DEBUG, "Skipped PREFIX SEI %d\n", payload_type);
35.         } else {
36.             av_log(s->avctx, AV_LOG_DEBUG, "Skipped PREFIX SEI %d\n", payload_type);
37.             skip_bits(gb, 8*payload_size);
38.         }
39.     } else { /* nal_unit_type == NAL_SEI_SUFFIX */
40.         if (payload_type == 132 /* && s->decode_checksum_sei */)
41.             decode_nal_sei_decoded_picture_hash(s);
42.         else {
43.             av_log(s->avctx, AV_LOG_DEBUG, "Skipped SUFFIX SEI %d\n", payload_type);
44.             skip_bits(gb, 8 * payload_size);
45.         }
46.     }
47.     return 1;
48. }
```

从源代码可以看出，decode\_nal\_sei\_message()根据不同的payload\_type调用不同的函数进行处理，例如调用decode\_nal\_sei\_decoded\_picture\_hash(), decode\_nal\_sei\_frame\_packing\_arrangement(), decode\_nal\_sei\_display\_orientation()等等。

**雷霄骅**

leixiaohua1020@126.com

<http://blog.csdn.net/leixiaohua1020>

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/46412607>

文章标签： FFmpeg 解析 VPS SPS PPS

个人分类： FFMPEG

所属专栏： FFmpeg

此PDF由spygg生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com