

```
1  Changes from version 5.9 to version 6.0
2
3  Encoder:
4  - possible picture formats increased from (QCIF,CIF) to
5    (Sub-QCIF,QCIF,CIF,4CIF,16CIF,CUSTOM).
6    If CUSTOM is choosen, width and height must be devidable by 16
7    - all picture size dependent buffers are dynamically allocated in
8      get_mem4global_buffers() and partly in init_img()
9    - additional picture size parameters are added to config.dat
10
11  - re-structured code according to VCEG-M52.doc
12    "Test Model Document Changes for Data Partitioning and NAL Support"
13
14  - Context based adaptive binary arithmetic coding CABAC included
15    switch between UVLC and CABAC in config.dat
16    for further details see VCEG-L13.doc and VCEG-M59.doc
17
18  Decoder:
19  - re-structuring has just started, see section "known problems"
20
21
22  Known problems / open issues:
23
24  1 The current decoder doesn't work, as we have already started to
25    change the decoder.
26    Therefore the TML59 decoder binary is included as
27      \bin\ldecod59.exe.
28    This decoder is only tested for the following TML6 encoder mode
29    combination:
30      (UVLC && no data partitioning && no slices)
31
32  2 The picture header has to be changed to enable the coder to
33    signal the new picture formats
34    -> only QCIF and CIF format is decodable, results for other
35    formats only at
36    the encoder side available.
37
38  3 The encoder crashes for ((format > CIF) && (# of reference frames
39    != 5) && (B-frames))
40    this may be related to issue 8.
41
42  4 B frame mode is not very stable, there might be other mode
43    combinations including
44    B-frames that don't work.
45
46  5 In VCEG-M52.doc the following changes are suggested, but up to
47    now haven't been
48    incorporated in the software.
49    "The implicitly given decoding routines in the encoder will be
50    identified and modularized.
51    By transferring these routines to the decoder software project a
52    more consistent
53    realization of the encoder and decoder is achieved.
54    This will be important for future integration processes since it
55    may reduce the risk of
56    encoder/decoder mismatch and, at the same time, limit the burden
57    of a duplicated implementation."
58
59  6 The introduced modular software structure is implemented from the
60    top level down to
61    the macroblock level. It should be considered to have a modular
62    structure for the block
63    level as well.
64
65  7 Some of the global buffers may not be necessary, e.g. different
66    temp buffers
```

52 used in different functions may be merged to a single buffer.
53
54 8 It would be better to use just one buffer for all reference
frames and not one
55 additional for B-frame calculation (mref[no_multpred+1] instead
mref[no_multpred]
56 and mref_P.
57
58 9 Other color formats than 4:2:0 and color depths than 8 bit/pel
and component should
59 be supported. Related input parameters are included in
config.dat, but currently
60 have no effect on the encoding process.
61
62 Contact: heising@hhi.de
63 marpe@hhi.de
64 stockhammer@ei.tum.de

```
1  /*****
2  **
3  *  image.h for H.26L encoder.
4  *  Copyright (C) 1999  Telenor Satellite Services, Norway
5  *
6  *  Contacts:
7  *  Inge Lille-Langøy          <inge.lille-langoy@telenor.com>
8  *
9  *
10 *  Telenor Satellite Services
11 *  P.O.Box 6914 St.Olavs plass
12 *  N-0130 Oslo, Norway
13 *
14 *****/
15 #ifndef _IMAGE_H_
16 #define _IMAGE_H_
17
18 /* QP dependent filter strenght clipping for loopfilter */
19 const byte FILTER_STR[32][4] =
20 {
21     {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},
22     },{0,0,0,0},
23     {0,0,0,0},{0,0,0,1},{0,0,0,1},{0,0,1,1},{0,0,1,1},{0,0,1,1},{0,1,1,1},
24     },{0,1,1,1},
25     {0,1,1,1},{0,1,1,2},{0,1,1,2},{0,1,1,2},{0,1,1,2},{0,1,2,3},{0,1,2,3},
26     },{0,1,2,3},
27     {0,1,2,4},{0,2,3,4},{0,2,3,5},{0,2,3,5},{0,2,3,5},{0,2,4,7},{0,3,5,8},
28     },{0,3,5,9},
29 };
30
31 /* TAPs used in the oneforthpix() routine */
32 const int ONE_FOURTH_TAP[3][2] =
33 {
34     {20,20},
35     {-5,-4},
36     { 1, 0},
37 };
38
39 const byte LIM[32] =
40 {
41     7,  8,  9, 10, 11, 12, 14, 16,
42     18, 20, 22, 25, 28, 31, 35, 39,
43     44, 49, 55, 62, 69, 78, 88, 98,
44     110,124,139,156,175,197,221,248
45 };
46 #endif
```

```

1  //
   *****
   *****
2  //
   *****
   *****
3  // Block.c    Process one block
4  //
5  // Main contributors (see contributors.h for copyright, address and
   affiliation details)
6  //
7  // Inge Lille-Langøy          <inge.lille-langoy@telenor.com>
8  // Rickard Sjöberg           <rickard.sjoberg@era.ericsson.se>
9  // Stephan Wenger            <stewe@cs.tu-berlin.de>
10 // Jani Lainema              <jani.lainema@nokia.com>
11 // Detlev Marpe              <marpe@hhi.de>
12 //
   *****
   *****
13 //
   *****
   *****
14 #include "contributors.h"
15
16
17 #include <math.h>
18 #include <stdlib.h>
19 #include <stdio.h>
20
21 #include "global.h"
22 #include "block.h"
23
24 /*****
   **
25  *
26  *  Routine      : intrapred_luma()
27  *
28  *  Description: Make intra 4x4 prediction according to all 6
   prediction modes.
29  *              The routine uses left and upper neighbouring points
   from
30  *              previous coded blocks to do this (if available).
   Notice that
31  *              inaccessible neighbouring points are signalled with a
   negative
32  *              value i the predmode array .
33  *
34  *  Input       : Starting point of current 4x4 block image posision
35  *
36  *  Output      : none
37  *
38  *****/
39 void intrapred_luma(struct img_par *img,int img_x,int img_y)
40 {
41     int i,j,s0,s1,s2,ia[7][3],s[4][2];
42
43     int block_available_up = (img->ipredmode[img_x/BLOCK_SIZE+1][img_y
   /BLOCK_SIZE] >=0);
44     int block_available_left = (img->ipredmode[img_x/BLOCK_SIZE][img_y
   /BLOCK_SIZE+1] >=0);
45
46     s1=0;
47     s2=0;
48
49     /* make DC prediction */

```

```

50     for (i=0; i < BLOCK_SIZE; i++)
51     {
52         if (block_available_up)
53             s1 += imgY[img_y-1][img_x+i];    /* sum hor pix */
54         if (block_available_left)
55             s2 += imgY[img_y+i][img_x-1];    /* sum vert pix */
56     }
57     if (block_available_up && block_available_left)
58         s0=(s1+s2+4)/(2*BLOCK_SIZE);        /* no edge */
59     if (!block_available_up && block_available_left)
60         s0=(s2+2)/BLOCK_SIZE;                /* upper edge */
61     if (block_available_up && !block_available_left)
62         s0=(s1+2)/BLOCK_SIZE;                /* left edge */
63     if (!block_available_up && !block_available_left)
64         s0=128;                              /* top left corner, nothing
        to predict from */
65
66     for (i=0; i < BLOCK_SIZE; i++)
67     {
68         /* vertical prediction */
69         if (block_available_up)
70             s[i][0]=imgY[img_y-1][img_x+i];
71         /* horizontal prediction */
72         if (block_available_left)
73             s[i][1]=imgY[img_y+i][img_x-1];
74     }
75
76     for (j=0; j < BLOCK_SIZE; j++)
77     {
78         for (i=0; i < BLOCK_SIZE; i++)
79         {
80             img->mprrr[DC_PRED][i][j]=s0;    /* store DC prediction */
81             img->mprrr[VERT_PRED][i][j]=s[j][0]; /* store vertical
            prediction */
82             img->mprrr[HOR_PRED][i][j]=s[i][1]; /* store horizontal
            prediction */
83         }
84     }
85
86     /* Prediction according to 'diagonal' modes */
87     if (block_available_up && block_available_left)
88     {
89         int A = imgY[img_y-1][img_x];
90         int B = imgY[img_y-1][img_x+1];
91         int C = imgY[img_y-1][img_x+2];
92         int D = imgY[img_y-1][img_x+3];
93         int E = imgY[img_y ][img_x-1];
94         int F = imgY[img_y+1][img_x-1];
95         int G = imgY[img_y+2][img_x-1];
96         int H = imgY[img_y+3][img_x-1];
97         int I = imgY[img_y-1][img_x-1];
98         ia[0][0]=(H+2*G+F+2)/4;
99         ia[1][0]=(G+2*F+E+2)/4;
100        ia[2][0]=(F+2*E+I+2)/4;
101        ia[3][0]=(E+2*I+A+2)/4;
102        ia[4][0]=(I+2*A+B+2)/4;
103        ia[5][0]=(A+2*B+C+2)/4;
104        ia[6][0]=(B+2*C+D+2)/4;
105        for (i=0; i<4; i++)
106            for (j=0; j<4; j++)
107                img->mprrr[DIAG_PRED_LR_45][i][j]=ia[j-i+3][0];
108    }
109    if (block_available_up)
110    { /* Do prediction 1 */
111        int A = imgY[img_y-1][img_x+0];
112        int B = imgY[img_y-1][img_x+1];

```

```

113     int C = imgY[img_y-1][img_x+2];
114     int D = imgY[img_y-1][img_x+3];
115
116     img->mprrr[DIAG_PRED_RL][0][0] = (A+B)/2; /* a */
117     img->mprrr[DIAG_PRED_RL][1][0] = B;        /* e */
118     img->mprrr[DIAG_PRED_RL][0][1] = img->mprrr[DIAG_PRED_RL][2][0]
119     = (B+C)/2; /* b i */
120     img->mprrr[DIAG_PRED_RL][1][1] = img->mprrr[DIAG_PRED_RL][3][0]
121     = C;        /* f m */
122     img->mprrr[DIAG_PRED_RL][0][2] = img->mprrr[DIAG_PRED_RL][2][1]
123     = (C+D)/2; /* c j */
124     img->mprrr[DIAG_PRED_RL][3][1] =
125     img->mprrr[DIAG_PRED_RL][1][2] =
126     img->mprrr[DIAG_PRED_RL][2][2] =
127     img->mprrr[DIAG_PRED_RL][3][2] =
128     img->mprrr[DIAG_PRED_RL][0][3] =
129     img->mprrr[DIAG_PRED_RL][1][3] =
130     img->mprrr[DIAG_PRED_RL][2][3] =
131     img->mprrr[DIAG_PRED_RL][3][3] = D;
132     /* d g h k l n o p */
133 }
134
135 if (block_available_left)
136 { /* Do prediction 5 */
137     int E = imgY[img_y+0][img_x-1];
138     int F = imgY[img_y+1][img_x-1];
139     int G = imgY[img_y+2][img_x-1];
140     int H = imgY[img_y+3][img_x-1];
141
142     img->mprrr[DIAG_PRED_LR][0][0] = (E+F)/2; /* a */
143     img->mprrr[DIAG_PRED_LR][0][1] = F;        /* b */
144     img->mprrr[DIAG_PRED_LR][1][0] = img->mprrr[DIAG_PRED_LR][0][2]
145     = (F+G)/2; /* e c */
146     img->mprrr[DIAG_PRED_LR][1][1] = img->mprrr[DIAG_PRED_LR][0][3]
147     = G;        /* f d */
148     img->mprrr[DIAG_PRED_LR][2][0] = img->mprrr[DIAG_PRED_LR][1][2]
149     = (G+H)/2; /* i g */
150
151     img->mprrr[DIAG_PRED_LR][1][3] =
152     img->mprrr[DIAG_PRED_LR][2][1] =
153     img->mprrr[DIAG_PRED_LR][2][2] =
154     img->mprrr[DIAG_PRED_LR][2][3] =
155     img->mprrr[DIAG_PRED_LR][3][0] =
156     img->mprrr[DIAG_PRED_LR][3][1] =
157     img->mprrr[DIAG_PRED_LR][3][2] =
158     img->mprrr[DIAG_PRED_LR][3][3] = H;
159 }
160 }
161
162 /*****
163 **
164 * Routine      : intrapred_luma_2()
165 * Description: 16x16 based luma prediction
166 * Input       : Image parameters
167 * Output      : none
168 **
169 *****/
170 void intrapred_luma_2(struct img_par *img)
171 {
172     int s0,s1,s2;
173     int i,j;

```

```

170     int s[16][2];
171
172     int ih,iv;
173     int ib,ic,iaa;
174
175     int mb_nr = img->current_mb_nr;
176     int mb_width = img->width/16;
177     int mb_available_up = (img->mb_y == 0) ? 0 : (img->slice_numbers[
mb_nr] == img->slice_numbers[mb_nr-mb_width]);
178     int mb_available_left = (img->mb_x == 0) ? 0 : (img->slice_numbers
[mb_nr] == img->slice_numbers[mb_nr-1]);
179
180     s1=s2=0;
181     /* make DC prediction */
182     for (i=0; i < MB_BLOCK_SIZE; i++)
183     {
184         if (mb_available_up)
185             s1 += imgY[img->pix_y-1][img->pix_x+i];    /* sum hor
pix */
186         if (mb_available_left)
187             s2 += imgY[img->pix_y+i][img->pix_x-1];    /* sum vert
pix */
188     }
189     if (mb_available_up && mb_available_left)
190         s0=(s1+s2+16)/(2*MB_BLOCK_SIZE);    /* no
edge */
191     if (!mb_available_up && mb_available_left)
192         s0=(s2+8)/MB_BLOCK_SIZE;    /* upper
edge */
193     if (mb_available_up && !mb_available_left)
194         s0=(s1+8)/MB_BLOCK_SIZE;    /* left
edge */
195     if (!mb_available_up && !mb_available_left)
196         s0=128;    /* top left
corner, nothing to predict from */
197
198     for (i=0; i < MB_BLOCK_SIZE; i++)
199     {
200         /* vertical prediction */
201         if (mb_available_up)
202             s[i][0]=imgY[img->pix_y-1][img->pix_x+i];
203         /* horizontal prediction */
204         if (mb_available_left)
205             s[i][1]=imgY[img->pix_y+i][img->pix_x-1];
206     }
207
208     for (j=0; j < MB_BLOCK_SIZE; j++)
209     {
210         for (i=0; i < MB_BLOCK_SIZE; i++)
211         {
212             img->mprrr_2[VERT_PRED_16][j][i]=s[i][0]; /* store
vertical prediction */
213             img->mprrr_2[HOR_PRED_16][j][i]=s[j][1]; /* store
horizontal prediction */
214             img->mprrr_2[DC_PRED_16][j][i]=s0;    /* store DC
prediction */
215         }
216     }
217     if (!mb_available_up || !mb_available_left) /* edge */
218         return;
219
220     /* 16 bit integer plan pred */
221
222     ih=0;
223     iv=0;
224     for (i=1;i<9;i++)

```

```

225     {
226         ih += i*(imgY[img->pix_y-1][img->pix_x+7+i] - imgY[img->pix_y-
1][img->pix_x+7-i]);
227         iv += i*(imgY[img->pix_y+7+i][img->pix_x-1] - imgY[img->pix_y+
7-i][img->pix_x-1]);
228     }
229     ib=5*(ih/4)/16;
230     ic=5*(iv/4)/16;
231
232     iaa=16*(imgY[img->pix_y-1][img->pix_x+15]+imgY[img->pix_y+15][img
->pix_x-1]);
233     for (j=0;j< MB_BLOCK_SIZE;j++)
234     {
235         for (i=0;i< MB_BLOCK_SIZE;i++)
236         {
237             img->mprrr_2[PLANE_16][j][i]=(iaa+(i-7)*ib +(j-7)*ic + 16)/
32;/* store plane prediction */
238         }
239     }
240 }
241
242 /*****
243 *
244 * Routine      : find_sad2()
245 *
246 * Description: Find best 16x16 based intra mode
247 *
248 *
249 * Input       : Image parameters, pointer to best 16x16 intra mode
250 *
251 * Output      : best 16x16 based SAD
252 *
253 *****/
254 int find_sad2(struct img_par *img,int *intra_mode)
255 {
256     int current_intra_sad_2,best_intra_sad2;
257     int M1[16][16],M0[4][4][4][4],M3[4],M4[4][4];
258
259     int i,j,k;
260     int ii,jj;
261
262     best_intra_sad2=MAX_VALUE;
263
264     for (k=0;k<4;k++)
265     {
266         int mb_nr = img->current_mb_nr;
267         int mb_width = img->width/16;
268         int mb_available_up = (img->mb_y == 0) ? 0 : (img->
slice_numbers[mb_nr] == img->slice_numbers[mb_nr-mb_width]);
269         int mb_available_left = (img->mb_x == 0) ? 0 : (img->
slice_numbers[mb_nr] == img->slice_numbers[mb_nr-1]);
270
271         /*check if there are neighbours to predict from */
272         if ((k==0 && !mb_available_up) || (k==1 && !mb_available_left)
|| (k==3 && (!mb_available_left || !mb_available_up)))
273         {
274             ; /* edge, do nothing */
275         }
276         else
277         {
278             for (j=0;j<16;j++)
279             {
280                 for (i=0;i<16;i++)
281                 {

```

```

282         M1[i][j]=imgY_org[img->pix_y+j][img->pix_x+i]-img
        ->mprrr_2[k][j][i];
283         M0[i%4][i/4][j%4][j/4]=M1[i][j];
284     }
285 }
286 current_intra_sad_2=0;          /* no SAD start
handicap here */
287 for (jj=0;jj<4;jj++)
288 {
289     for (ii=0;ii<4;ii++)
290     {
291         for (j=0;j<4;j++)
292         {
293             M3[0]=M0[0][ii][j][jj]+M0[3][ii][j][jj];
294             M3[1]=M0[1][ii][j][jj]+M0[2][ii][j][jj];
295             M3[2]=M0[1][ii][j][jj]-M0[2][ii][j][jj];
296             M3[3]=M0[0][ii][j][jj]-M0[3][ii][j][jj];
297
298             M0[0][ii][j][jj]=M3[0]+M3[1];
299             M0[2][ii][j][jj]=M3[0]-M3[1];
300             M0[1][ii][j][jj]=M3[2]+M3[3];
301             M0[3][ii][j][jj]=M3[3]-M3[2];
302         }
303
304         for (i=0;i<4;i++)
305         {
306             M3[0]=M0[i][ii][0][jj]+M0[i][ii][3][jj];
307             M3[1]=M0[i][ii][1][jj]+M0[i][ii][2][jj];
308             M3[2]=M0[i][ii][1][jj]-M0[i][ii][2][jj];
309             M3[3]=M0[i][ii][0][jj]-M0[i][ii][3][jj];
310
311             M0[i][ii][0][jj]=M3[0]+M3[1];
312             M0[i][ii][2][jj]=M3[0]-M3[1];
313             M0[i][ii][1][jj]=M3[2]+M3[3];
314             M0[i][ii][3][jj]=M3[3]-M3[2];
315             for (j=0;j<4;j++)
316                 if ((i+j)!=0)
317                     current_intra_sad_2 += abs(M0[i][ii][j
318                                     ][jj]);
319         }
320     }
321
322     for (j=0;j<4;j++)
323         for (i=0;i<4;i++)
324             M4[i][j]=M0[0][i][0][j]/4;
325
326     /* Hadamard of DC koeff */
327     for (j=0;j<4;j++)
328     {
329         M3[0]=M4[0][j]+M4[3][j];
330         M3[1]=M4[1][j]+M4[2][j];
331         M3[2]=M4[1][j]-M4[2][j];
332         M3[3]=M4[0][j]-M4[3][j];
333
334         M4[0][j]=M3[0]+M3[1];
335         M4[2][j]=M3[0]-M3[1];
336         M4[1][j]=M3[2]+M3[3];
337         M4[3][j]=M3[3]-M3[2];
338     }
339
340     for (i=0;i<4;i++)
341     {
342         M3[0]=M4[i][0]+M4[i][3];
343         M3[1]=M4[i][1]+M4[i][2];
344         M3[2]=M4[i][1]-M4[i][2];

```

```

345         M3[3]=M4[i][0]-M4[i][3];
346
347         M4[i][0]=M3[0]+M3[1];
348         M4[i][2]=M3[0]-M3[1];
349         M4[i][1]=M3[2]+M3[3];
350         M4[i][3]=M3[3]-M3[2];
351
352         for (j=0;j<4;j++)
353             current_intra_sad_2 += abs(M4[i][j]);
354     }
355     if(current_intra_sad_2 < best_intra_sad2)
356     {
357         best_intra_sad2=current_intra_sad_2;
358         *intra_mode = k; /* update best intra mode */
359     }
360 }
361 }
362 }
363 best_intra_sad2 = best_intra_sad2/2;
364
365 return best_intra_sad2;
366
367 }
368 /*****
369 *
370 * Routine:      dct_luma2()
371 *
372 * Description:  For new intra pred routines
373 *
374 *
375 * Input:       Image par, 16x16 based intra mode
376 *
377 * Output:
378 *
379 *****/
380 void dct_luma2(struct img_par *img,int new_intra_mode)
381 {
382     int jq0;
383     int i,j;
384     int ii,jj;
385     int il,jl;
386
387     int M1[16][16];
388     int M4[4][4];
389     int M5[4],M6[4];
390     int M0[4][4][4][4];
391     int coeff[16];
392
393     int quant_set,run,scan_pos,coeff_ctr,level;
394
395     jq0=JQQ3;
396
397     for (j=0;j<16;j++)
398     {
399         for (i=0;i<16;i++)
400         {
401             M1[i][j]=imgY_org[img->pix_y+j][img->pix_x+i]-img->mprrr_2[
new_intra_mode][j][i];
402             M0[i%4][i/4][j%4][j/4]=M1[i][j];
403         }
404     }
405
406     for (jj=0;jj<4;jj++)
407     {

```

```

408         for (ii=0;ii<4;ii++)
409         {
410             for (j=0;j<4;j++)
411             {
412                 for (i=0;i<2;i++)
413                 {
414                     i1=3-i;
415                     M5[i]= M0[i][ii][j][jj]+M0[i1][ii][j][jj];
416                     M5[i1]= M0[i][ii][j][jj]-M0[i1][ii][j][jj];
417                 }
418                 M0[0][ii][j][jj]=(M5[0]+M5[1])*13;
419                 M0[2][ii][j][jj]=(M5[0]-M5[1])*13;
420                 M0[1][ii][j][jj]=M5[3]*17+M5[2]*7;
421                 M0[3][ii][j][jj]=M5[3]*7-M5[2]*17;
422             }
423             /* vertical */
424             for (i=0;i<4;i++)
425             {
426                 for (j=0;j<2;j++)
427                 {
428                     j1=3-j;
429                     M5[j] = M0[i][ii][j][jj]+M0[i][ii][j1][jj];
430                     M5[j1]= M0[i][ii][j][jj]-M0[i][ii][j1][jj];
431                 }
432                 M0[i][ii][0][jj]=(M5[0]+M5[1])*13;
433                 M0[i][ii][2][jj]=(M5[0]-M5[1])*13;
434                 M0[i][ii][1][jj]= M5[3]*17+M5[2]*7;
435                 M0[i][ii][3][jj]= M5[3]*7 -M5[2]*17;
436             }
437         }
438     }
439
440     /* pick out DC coeff */
441
442     for (j=0;j<4;j++)
443         for (i=0;i<4;i++)
444             M4[i][j]= 49 * M0[0][i][0][j]/32768;
445
446     for (j=0;j<4;j++)
447     {
448         for (i=0;i<2;i++)
449         {
450             i1=3-i;
451             M5[i]= M4[i][j]+M4[i1][j];
452             M5[i1]=M4[i][j]-M4[i1][j];
453         }
454         M4[0][j]=(M5[0]+M5[1])*13;
455         M4[2][j]=(M5[0]-M5[1])*13;
456         M4[1][j]= M5[3]*17+M5[2]*7;
457         M4[3][j]= M5[3]*7 -M5[2]*17;
458     }
459
460     /* vertical */
461
462     for (i=0;i<4;i++)
463     {
464         for (j=0;j<2;j++)
465         {
466             j1=3-j;
467             M5[j]= M4[i][j]+M4[i][j1];
468             M5[j1]=M4[i][j]-M4[i][j1];
469         }
470         M4[i][0]=(M5[0]+M5[1])*13;
471         M4[i][2]=(M5[0]-M5[1])*13;
472         M4[i][1]= M5[3]*17+M5[2]*7;
473         M4[i][3]= M5[3]*7 -M5[2]*17;

```

```

474     }
475
476     /* quant , we use RD quant here as well*/
477
478     quant_set=img->qp;
479     run=-1;
480     scan_pos=0;
481     for (coeff_ctr=0;coeff_ctr<16;coeff_ctr++)
482     {
483         i=SNGL_SCAN[coeff_ctr][0];
484         j=SNGL_SCAN[coeff_ctr][1];
485         coeff[coeff_ctr]=M4[i][j];
486     }
487     rd_quant(img,QUANT_LUMA_SNG,coeff);
488
489     for (coeff_ctr=0;coeff_ctr<16;coeff_ctr++)
490     {
491         i=SNGL_SCAN[coeff_ctr][0];
492         j=SNGL_SCAN[coeff_ctr][1];
493
494         run++;
495
496         level=abs(coeff[coeff_ctr]);
497
498         if (level != 0)
499         {
500             img->cof[0][0][scan_pos][0][1]=sign(level,M4[i][j]);
501             img->cof[0][0][scan_pos][1][1]=run;
502             ++scan_pos;
503             run=-1;
504         }
505         M4[i][j]=sign(level,M4[i][j]);
506     }
507     img->cof[0][0][scan_pos][0][1]=0;
508
509     /* invers DC transform */
510
511     for (j=0;j<4;j++)
512     {
513         for (i=0;i<4;i++)
514             M5[i]=M4[i][j];
515
516         M6[0]=(M5[0]+M5[2])*13;
517         M6[1]=(M5[0]-M5[2])*13;
518         M6[2]= M5[1]*7 -M5[3]*17;
519         M6[3]= M5[1]*17+M5[3]*7;
520
521         for (i=0;i<2;i++)
522         {
523             i1=3-i;
524             M4[i][j]= M6[i]+M6[i1];
525             M4[i1][j]=M6[i]-M6[i1];
526         }
527     }
528
529     for (i=0;i<4;i++)
530     {
531         for (j=0;j<4;j++)
532             M5[j]=M4[i][j];
533
534         M6[0]=(M5[0]+M5[2])*13;
535         M6[1]=(M5[0]-M5[2])*13;
536         M6[2]= M5[1]*7 -M5[3]*17;
537         M6[3]= M5[1]*17+M5[3]*7;
538
539         for (j=0;j<2;j++)

```

```

540     {
541         j1=3-j;
542         M0[0][i][0][j] = ((M6[j]+M6[j1])/8) *JQ[quant_set][1];
543         M0[0][i][0][j1]= ((M6[j]-M6[j1])/8) *JQ[quant_set][1];
544     }
545 }
546 for (j=0;j<4;j++)
547 {
548     for (i=0;i<4;i++)
549     {
550         M0[0][i][0][j] = 3 * M0[0][i][0][j]/256;
551     }
552 }
553
554 /* AC invers trans/quant for MB */
555 img->kac=0;
556 for (jj=0;jj<4;jj++)
557 {
558     for (ii=0;ii<4;ii++)
559     {
560         run=-1;
561         scan_pos=0;
562         for (coeff_ctr=1;coeff_ctr<16;coeff_ctr++) /* set in AC
563             coeff */
564         {
565             i=SNGL_SCAN[coeff_ctr][0];
566             j=SNGL_SCAN[coeff_ctr][1];
567             coeff[coeff_ctr-1]=M0[i][ii][j][jj];
568         }
569         rd_quant(img,QUANT_LUMA_AC,coeff);
570
571         for (coeff_ctr=1;coeff_ctr<16;coeff_ctr++) /* set in AC
572             coeff */
573         {
574             i=SNGL_SCAN[coeff_ctr][0];
575             j=SNGL_SCAN[coeff_ctr][1];
576             run++;
577
578             level=abs(coeff[coeff_ctr-1]);
579
580             if (level != 0)
581             {
582                 img->kac=1;
583                 img->cof[ii][jj][scan_pos][0][0]=sign(level,M0[i][
584                     ii][j][jj]);
585                 img->cof[ii][jj][scan_pos][1][0]=run;
586                 ++scan_pos;
587                 run=-1;
588             }
589             M0[i][ii][j][jj]=sign(level*JQ[quant_set][1],M0[i][ii
590                 ][j][jj]);
591         }
592         img->cof[ii][jj][scan_pos][0][0]=0;
593
594         /* IDCT horizontal */
595
596         for (j=0;j<4;j++)
597         {
598             for (i=0;i<4;i++)
599             {
600                 M5[i]=M0[i][ii][j][jj];
601             }

```

```

602         M6[3]=M5[1]*17+M5[3]*7;
603
604         for (i=0;i<2;i++)
605         {
606             i1=3-i;
607             M0[i][ii][j][jj] =M6[i]+M6[i1];
608             M0[i1][ii][j][jj]=M6[i]-M6[i1];
609         }
610     }
611
612     /* vert */
613     for (i=0;i<4;i++)
614     {
615         for (j=0;j<4;j++)
616             M5[j]=M0[i][ii][j][jj];
617
618         M6[0]=(M5[0]+M5[2])*13;
619         M6[1]=(M5[0]-M5[2])*13;
620         M6[2]=M5[1]*7 -M5[3]*17;
621         M6[3]=M5[1]*17+M5[3]*7;
622
623         for (j=0;j<2;j++)
624         {
625             j1=3-j;
626             M0[i][ii][ j][jj]=M6[j]+M6[j1];
627             M0[i][ii][j1][jj]=M6[j]-M6[j1];
628
629         }
630     }
631
632 }
633
634
635 for (j=0;j<16;j++)
636 {
637     for (i=0;i<16;i++)
638     {
639         M1[i][j]=M0[i%4][i/4][j%4][j/4];
640     }
641 }
642
643 for (j=0;j<16;j++)
644     for (i=0;i<16;i++)
645         imgY[img->pix_y+j][img->pix_x+i]=min(255,max(0,(M1[i][j]+
646             img->mprrr_2[new_intra_mode][j][i]*JQQ1+JQQ2)/JQQ1));
647
648
649
650
651
652 /*****
653  *
654  *   Routine:      intrapred_chroma()
655  *
656  *   Description:  Intra prediction for chroma.  There is only one
657  *                 prediction mode,
658  *                 corresponding to 'DC prediction' for luma.
659  *                 However,since 2x2 transform
660  *                 of DC levels are used,all predictions are made from
661  *                 neighbouring MBs.
662  *                 Prediction also depends on whether the block is at a
663  *                 frame edge.
664  *
665  *   Input:        Starting point of current chroma macro block image

```

```

position
662 *
663 *   Output:      8x8 array with DC intra chroma prediction and diff
array
664 *
665 *****
*/

666
667 void intrapred_chroma(struct img_par *img,int img_c_x,int img_c_y,int
uv)
668 {
669     int s[2][2],s0,s1,s2,s3;
670     int i,j;
671
672     int mb_nr = img->current_mb_nr;
673     int mb_width = img->width/16;
674     int mb_available_up = (img_c_y/BLOCK_SIZE == 0) ? 0 : (img->
slice_numbers[mb_nr] == img->slice_numbers[mb_nr-mb_width]);
675     int mb_available_left = (img_c_x/BLOCK_SIZE == 0) ? 0 : (img->
slice_numbers[mb_nr] == img->slice_numbers[mb_nr-1]);
676
677     s0=s1=s2=s3=0;          /* reset counters */
678
679     for (i=0; i < BLOCK_SIZE; i++)
680     {
681         if(mb_available_up)
682         {
683             s0 += imgUV[uv][img_c_y-1][img_c_x+i];
684             s1 += imgUV[uv][img_c_y-1][img_c_x+i+BLOCK_SIZE];
685         }
686         if(mb_available_left)
687         {
688             s2 += imgUV[uv][img_c_y+i][img_c_x-1];
689             s3 += imgUV[uv][img_c_y+i+BLOCK_SIZE][img_c_x-1];
690         }
691     }
692
693     if(mb_available_up && mb_available_left)
694     {
695         s[0][0]=(s0+s2+4)/(2*BLOCK_SIZE);
696         s[1][0]=(s1+2)/BLOCK_SIZE;
697         s[0][1]=(s3+2)/BLOCK_SIZE;
698         s[1][1]=(s1+s3+4)/(2*BLOCK_SIZE);
699     }
700     else
701         if(mb_available_up && !mb_available_left)
702         {
703             s[0][0]=(s0+2)/BLOCK_SIZE;
704             s[1][0]=(s1+2)/BLOCK_SIZE;
705             s[0][1]=(s0+2)/BLOCK_SIZE;
706             s[1][1]=(s1+2)/BLOCK_SIZE;
707         }
708     else
709         if(!mb_available_up && mb_available_left)
710         {
711             s[0][0]=(s2+2)/BLOCK_SIZE;
712             s[1][0]=(s2+2)/BLOCK_SIZE;
713             s[0][1]=(s3+2)/BLOCK_SIZE;
714             s[1][1]=(s3+2)/BLOCK_SIZE;
715         }
716     else
717         if(!mb_available_up && !mb_available_left)
718         {
719             s[0][0]=128;
720             s[1][0]=128;
721             s[0][1]=128;

```

```

722             s[1][1]=128;
723         }
724     for (j=0; j < MB_BLOCK_SIZE/2; j++)
725     {
726         for (i=0; i < MB_BLOCK_SIZE/2; i++)
727         {
728             img->mpr[i][j]=s[i/BLOCK_SIZE][j/BLOCK_SIZE];
729             img->m7[i][j]=imgUV_org[uv][img_c_y+j][img_c_x+i]-img->mpr
                [i][j];
730         }
731     }
732 }
733
734 /*****
735  *
736  *  Routine:      motion_search()
737  *
738  *  Description: In this routine motion search (integer pel+1/3 pel)
and mode selection
739  *                is performed. Since we treat all 4x4 blocks before
coding/decoding the
740  *                prediction may not be based on decoded pixels (except
for some of the blocks).
741  *                This will result in too good prediction. To
compensate for this the SAD for
742  *                intra(tot_intra_sad) is given a 'handicap' depending
on QP.
743  *
744  *  Input:        Best intra SAD value.
745  *
746  *  Output:       Reference image.
747  *
748  *****/
749 int motion_search(struct inp_par *inp_par, struct img_par *img, int
tot_intra_sad)
750 {
751     int predframe_no;
752     int numc, j, ref_frame, i;
753     int vec1_x, vec1_y, vec2_x, vec2_y;
754     int pic_block_x, pic_block_y, block_x, ref_inx, block_y, pic_pix_y,
pic4_pix_y, pic_pix_x, pic4_pix_x, i22, lv, hv;
755     int all_mv[4][4][MAX_MULT_PRED][9][2];
756     int center_x, s_pos_x, ip0, ip1, center_y, blocktype, s_pos_y, iy0, jy0,
center_h2, curr_search_range, center_v2;
757     int s_pos_x1, s_pos_y1, s_pos_x2, s_pos_y2;
758     int abort_search;
759     int mb_y, mb_x;
760     int min_inter_sad, tot_inter_sad, best_inter_sad, current_inter_sad;
761     int mo[16][16];
762     int blockshape_x, blockshape_y;
763     int tmp0, tmp1;
764     int mb_nr = img->current_mb_nr;
765     int mb_width = img->width/16;
766     int mb_available_up    = (img->mb_y == 0) ? 0 : (img->
slice_numbers[mb_nr] == img->slice_numbers[mb_nr-mb_width]);
767     int mb_available_left  = (img->mb_x == 0) ? 0 : (img->
slice_numbers[mb_nr] == img->slice_numbers[mb_nr-1]);
768     int mb_available_upleft = (img->mb_x == 0 || img->mb_y == 0) ? 0
: (img->slice_numbers[mb_nr] == img->slice_numbers[mb_nr-mb_width-
1]);
769     int mb_available_upright = (img->mb_x >= mb_width-1 || img->mb_y
== 0) ? 0 : (img->slice_numbers[mb_nr] == img->slice_numbers[mb_nr
-mb_width+1]);
770     int block_available_up, block_available_left,

```

```

    block_available_upright, block_available_upleft;
771     int mv_a, mv_b, mv_c, mv_d, pred_vec;
772     int mvPredType, rFrameL, rFrameU, rFrameUR;
773
774     min_inter_sad=MAX_VALUE; /* start value for inter SAD search */
775
776     /* Loop through all reference frames under consideration */
777     for (ref_frame=0; ref_frame < min(img->number,inp_par->no_multpred
    ); ref_frame++)
778     {
779
780         for (j = 0;j < 4;j++)
781         {
782             for (i = 0;i < 4;i++)
783             {
784                 refFrArr[img->block_y+j][img->block_x+i] = ref_frame;
785             }
786         }
787
788         /* find reference image */
789         ref_inx=(img->number-ref_frame-1)%inp_par->no_multpred; /*GH
    inp->no_multpred used insteadof MAX_MULT_PRED
790
    frame buffer size = inp->no_multpred+1*/
791
792         /* Looping through all the chosen block sizes: */
793         blocktype=1;
794         while (inp_par->bkc_size[blocktype][0]==0 && blocktype<=7) /*
    skip block sizes not chosen */
795             blocktype++;
796
797         for (;blocktype <= 7;)
798         {
799             blockshape_x=inp_par->bkc_size[blocktype][0];/*
    inp_par->bkc_size has information of the 7 block sizes */
800             blockshape_y=inp_par->bkc_size[blocktype][1];/* array iz
    stores offset inside one MB */
801
802             /* curr_search_range is the actual search range used
    depending on block size and reference frame.
803             It may be reduced by 1/2 or 1/4 for smaller blocks and
    prediction from older frames due to complexity
804             */
805             curr_search_range=inp_par->search_range/((min(ref_frame,1
    )+1) * min(2,blocktype));
806
807             tot_inter_sad=QP2QUANT[img->qp] * min(ref_frame,1) * 2;
    /* start 'handicap ' */
808
809             /* Loop through the whole MB with all block sizes */
810             for (mb_y=0; mb_y < MB_BLOCK_SIZE; mb_y += blockshape_y)
811             {
812                 block_available_up = mb_available_up || (mb_y > 0);
813
814                 block_y=mb_y/BLOCK_SIZE;
815                 pic_block_y=img->block_y+block_y;
816                 pic_pix_y=img->pix_y+mb_y;
817                 pic4_pix_y=pic_pix_y*4;
818
819                 for (mb_x=0; mb_x < MB_BLOCK_SIZE; mb_x +=
    blockshape_x)
820                 {
821                     block_x=mb_x/BLOCK_SIZE;
822                     pic_block_x=img->block_x+block_x;
823                     pic_pix_x=img->pix_x+mb_x;
824                     pic4_pix_x=pic_pix_x*4;

```

```

825
826      /* D B C */
827      /* A X   */
828
829      /* 1 A, B, D are set to 0 if unavailable */
830      /* 2 If C is not available it is replaced by D */
831
832      block_available_left = mb_available_left || (mb_x
833      > 0);
834
835      if (mb_y > 0)
836          block_available_upright = mb_x+blockshape_x !=
837          MB_BLOCK_SIZE ? 1 : 0;
838      else if (mb_x+blockshape_x != MB_BLOCK_SIZE)
839          block_available_upright = block_available_up;
840      else
841          block_available_upright = mb_available_upright;
842
843      if (mb_x > 0)
844          block_available_upleft = mb_y > 0 ? 1 :
845          block_available_up;
846      else if (mb_y > 0)
847          block_available_upleft = block_available_left;
848      else
849          block_available_upleft = mb_available_upleft;
850
851      mvPredType = MVPRED_MEDIAN;
852
853      rFrameL    = block_available_left    ? refFrArr[
854      pic_block_y][pic_block_x-1] : -1;
855      rFrameU    = block_available_up      ? refFrArr[
856      pic_block_y-1][pic_block_x] : -1;
857      rFrameUR   = block_available_upright ? refFrArr[
858      pic_block_y-1][pic_block_x+blockshape_x/4] :
859      block_available_upleft ? refFrArr[
860      pic_block_y-1][pic_block_x-1] : -1;
861
862      /* Prediction if only one of the neighbors uses
863      the reference frame
864      * we are checking
865      */
866
867      if(rFrameL == ref_frame && rFrameU != ref_frame &&
868      rFrameUR != ref_frame)
869          mvPredType = MVPRED_L;
870      else if(rFrameL != ref_frame && rFrameU ==
871      ref_frame && rFrameUR != ref_frame)
872          mvPredType = MVPRED_U;
873      else if(rFrameL != ref_frame && rFrameU !=
874      ref_frame && rFrameUR == ref_frame)
875          mvPredType = MVPRED_UR;
876
877      /* Directional predictions */
878
879      else if(blockshape_x == 8 && blockshape_y == 16)
880      {
881          if(mb_x == 0)
882          {
883              if(rFrameL == ref_frame)
884                  mvPredType = MVPRED_L;
885          }
886          else
887          {
888              if(rFrameUR == ref_frame)
889                  mvPredType = MVPRED_UR;
890          }
891      }

```

```

880     }
881     else if(blockshape_x == 16 && blockshape_y == 8)
882     {
883         if(mb_y == 0)
884         {
885             if(rFrameU == ref_frame)
886                 mvPredType = MVPRED_U;
887         }
888         else
889         {
890             if(rFrameL == ref_frame)
891                 mvPredType = MVPRED_L;
892         }
893     }
894     else if(blockshape_x == 8 && blockshape_y == 4 &&
mb_x == 8)
895         mvPredType = MVPRED_L;
896     else if(blockshape_x == 4 && blockshape_y == 8 &&
mb_y == 8)
897         mvPredType = MVPRED_U;
898
899     for (hv=0; hv < 2; hv++)
900     {
901
902         mv_a = block_available_left    ? tmp_mv[hv][
pic_block_y][pic_block_x-1+4]    : 0;
903         mv_b = block_available_up      ? tmp_mv[hv][
pic_block_y-1][pic_block_x+4]    : 0;
904         mv_d = block_available_upleft  ? tmp_mv[hv][
pic_block_y-1][pic_block_x-1+4]  : 0;
905         mv_c = block_available_upright ? tmp_mv[hv][
pic_block_y-1][pic_block_x+blockshape_x/4+4] :
mv_d;
906
907         switch (mvPredType)
908         {
909
910             case MVPRED_MEDIAN:
911
912                 if(!(block_available_upleft ||
block_available_up ||
block_available_upright))
913                     pred_vec = mv_a;
914                 else
915                     pred_vec = mv_a+mv_b+mv_c-min(mv_a,min
(mv_b,mv_c))-max(mv_a,max(mv_b,mv_c));
916
917                     break;
918
919             case MVPRED_L:
920                 pred_vec = mv_a;
921                 break;
922             case MVPRED_U:
923                 pred_vec = mv_b;
924                 break;
925             case MVPRED_UR:
926                 pred_vec = mv_c;
927                 break;
928             default:
929                 break;
930         }
931
932         img->mv[block_x][block_y][ref_frame][blocktype
][hv] = pred_vec;
933     }
934

```

```

935     ip0=img->mv[block_x][block_y][ref_frame][blocktype
    ][0];
936     ip1=img->mv[block_x][block_y][ref_frame][blocktype
    ][1];
937
938     /* Integer pel search.  center_x,center_y is the
    'search center'*/
939
940     center_x=ip0/4;
941     center_y=ip1/4;
942
943     /* Limitation of center_x,center_y so that the
    search wlevel ow contains the (0,0) vector*/
944     center_x=max(-(inp_par->search_range),min(inp_par
    ->search_range,center_x));
945     center_y=max(-(inp_par->search_range),min(inp_par
    ->search_range,center_y));
946
947     /* Search center corrected to prevent vectors
    outside the frame, this is not permitted in this
    model. */
948
949     center_x=max(min(center_x,img->width-1-pic_pix_x-(
    blockshape_x-1)-curr_search_range),
    curr_search_range-pic_pix_x);
950     center_y=max(min(center_y,img->height_err-
    pic_pix_y-(blockshape_y-1)-curr_search_range),
    curr_search_range-pic_pix_y);
951
952     /* mo() is the the original block to be used in
    the search process */
953
954     for (i=0; i < blockshape_x; i++)
955         for (j=0; j < blockshape_y; j++)
956             mo[i][j]=imgY_org[pic_pix_y+j][pic_pix_x+i
    ];
957
958     best_inter_sad=MAX_VALUE;
959
960     numc=(curr_search_range*2+1)*(curr_search_range*2+
    1);
961
962     for (lv=0; lv < numc; lv++)
963     {
964         s_pos_x=(center_x+img->spiral_search[0][lv])*4;
965         s_pos_y=(center_y+img->spiral_search[1][lv])*4;
966
967         iy0=pic4_pix_x+s_pos_x;
968         jy0=pic4_pix_y+s_pos_y;
969
970         /*
971         The initial setting of current_inter_sad
          reflects the number of bits used to signal
          the vector.
972         It also depends on the QP. (See documented
          approach of RD constrained motion search)
          */
973         tmp0=s_pos_x-ip0;
974         tmp1=s_pos_y-ip1;
975         current_inter_sad=(QP2QUANT[img->qp]*(img->
          mv_bituse[absm(tmp0)]+img->mv_bituse[absm(tmp1
          )])));
976
977         if (s_pos_x == 0 && s_pos_y == 0 && blocktype
          == 1) /* no motion, 16x16 blocktype */
978         {
979

```

```

980         current_inter_sad -= QP2QUANT[img->qp] *
981         16;
982     }
983     abort_search=FALSE;
984     for (j=0; j < blockshape_y && !abort_search; j
985         ++){
986         {
987             vec2_y=jy0+j*4;
988
989             for (i=0; i < blockshape_x ; i++)
990             {
991                 tmp0=mo[i][j]-mref[ref_inx][vec2_y][
992                 iy0+i*4];
993
994                 current_inter_sad += absm(tmp0);
995             }
996             if (current_inter_sad >= best_inter_sad)
997             {
998                 abort_search=TRUE;
999             }
1000         }
1001         if(!abort_search)
1002         {
1003             center_h2=s_pos_x;
1004             center_v2=s_pos_y;
1005             best_inter_sad=current_inter_sad;
1006         }
1007     }
1008
1009     /*
1010     center_h2, center_v2 is the search center to be
1011     used for 1/2 pixel search.
1012     It is adjusted to prevent vectors pointing
1013     outside the frame.
1014     */
1015
1016     center_h2=max(2-pic4_pix_x,min(center_h2,(img->
1017     width-1-pic4_pix_x-(blockshape_x-1))*4-2));
1018     center_v2=max(2-pic4_pix_y,min(center_v2,(img->
1019     height_err-pic4_pix_y-(blockshape_y-1))*4-2));
1020
1021     /* 1/2 pixel search. In this version the search
1022     is over 9 vector positions. */
1023
1024     best_inter_sad=MAX_VALUE;
1025     for (lv=0; lv < 9; lv++)
1026     {
1027         s_pos_x=center_h2+img->spiral_search[0][lv]*2;
1028         s_pos_y=center_v2+img->spiral_search[1][lv]*2;
1029         iy0=pic4_pix_x+s_pos_x;
1030         jy0=pic4_pix_y+s_pos_y;
1031         tmp0=s_pos_x-ip0;
1032         tmp1=s_pos_y-ip1;
1033         current_inter_sad=(QP2QUANT[img->qp]*(img->
1034         mv_bituse[absm(tmp0)]+img->mv_bituse[absm(tmp1)
1035         ]))));
1036
1037         if (s_pos_x == 0 && s_pos_y == 0 && blocktype
1038         == 1) /* no motion, blocktype 16x16 */
1039         {
1040             current_inter_sad -= QP2QUANT[img->qp] *
1041             16;
1042         }
1043         for (vec1_x=0; vec1_x < blockshape_x; vec1_x
1044             += 4)

```

```

1033     {
1034         for (vec1_y=0; vec1_y < blockshape_y;
1035             vec1_y += 4)
1036         {
1037             for (i=0; i < BLOCK_SIZE; i++)
1038             {
1039                 vec2_x=i+vec1_x;
1040                 i22=iy0+vec2_x*4;
1041                 for (j=0; j < BLOCK_SIZE; j++)
1042                 {
1043                     vec2_y=j+vec1_y;
1044                     img->m7[i][j]=mo[vec2_x][
1045                         vec2_y]-mref[ref_inx][jy0+
1046                             vec2_y*4][i22];
1047                 }
1048             }
1049             current_inter_sad += find_sad(img,
1050                 inp_par->hadamard);
1051         }
1052     }
1053     if (current_inter_sad < best_inter_sad)
1054     {
1055         /* Vectors are saved in all_mv[] to be
1056            able to assign correct vectors to each
1057            block after mode selection.
1058            tmp_mv[] is a 'temporary' assignment
1059            of vectors to be used to estimate
1060            'bit cost' in vector prediction.
1061            */
1062         s_pos_x1=s_pos_x;
1063         s_pos_y1=s_pos_y;
1064         s_pos_x2=s_pos_x;
1065         s_pos_y2=s_pos_y;
1066         for (i=0; i < blockshape_x/BLOCK_SIZE; i++)
1067         {
1068             for (j=0; j < blockshape_y/BLOCK_SIZE;
1069                 j++)
1070             {
1071                 all_mv[block_x+i][block_y+j][
1072                     ref_frame][blocktype][0]=s_pos_x1;
1073                 all_mv[block_x+i][block_y+j][
1074                     ref_frame][blocktype][1]=s_pos_y1;
1075                 tmp_mv[0][pic_block_y+j][
1076                     pic_block_x+i+BLOCK_SIZE]=s_pos_x1;
1077                 tmp_mv[1][pic_block_y+j][
1078                     pic_block_x+i+BLOCK_SIZE]=s_pos_y1;
1079             }
1080         }
1081         best_inter_sad=current_inter_sad;
1082     }
1083 }
1084
1085 /* 1/4 pixel search. */
1086
1087 s_pos_x2=max(1-pic4_pix_x,min(s_pos_x2,(img->width
1088 -1-pic4_pix_x-(blockshape_x-1))*4-1));
1089 s_pos_y2=max(1-pic4_pix_y,min(s_pos_y2,(img->
1090 height_err-pic4_pix_y-(blockshape_y-1))*4-1));
1091 for (lv=1; lv < 9; lv++)
1092 {
1093     s_pos_x=s_pos_x2+img->spiral_search[0][lv];
1094     s_pos_y=s_pos_y2+img->spiral_search[1][lv];
1095     iy0=pic4_pix_x+s_pos_x;

```

```

1084     jy0=pic4_pix_y+s_pos_y;
1085     tmp0=s_pos_x-ip0;
1086     tmp1=s_pos_y-ip1;
1087     current_inter_sad=(QP2QUANT[img->qp]*(img->
mv_bituse[absm(tmp0)]+img->mv_bituse[absm(tmp1
))));

1088
1089     if (s_pos_x == 0 && s_pos_y == 0 && blocktype
== 1) /* no motion, blocktype 16x16 */
1090     {
1091         current_inter_sad -= QP2QUANT[img->qp] *
16;
1092     }
1093     for (vec1_x=0; vec1_x < blockshape_x; vec1_x
+= 4)
1094     {
1095         for (vec1_y=0; vec1_y < blockshape_y;
vec1_y += 4)
1096         {
1097             for (i=0; i < BLOCK_SIZE; i++)
1098             {
1099                 vec2_x=i+vec1_x;
1100                 i22=iy0+vec2_x*4;
1101                 for (j=0; j < BLOCK_SIZE; j++)
1102                 {
1103                     vec2_y=j+vec1_y;
1104                     img->m7[i][j]=mo[vec2_x][
vec2_y]-mref[ref_inx][jy0+
vec2_y*4][i22];
1105                 }
1106             }
1107             current_inter_sad += find_sad(img,
inp_par->hadamard);
1108         }
1109     }
1110     if (current_inter_sad < best_inter_sad)
1111     {
1112         /* Vectors are saved in all_mv[] to be
able to assign correct vectors to each
block after mode selection.
1113         tmp_mv[] is a 'temporary' assignment
of vectors to be used to estimate
'bit cost' in vector prediction.
1114         */
1115
1116         s_pos_x1=s_pos_x;
1117         s_pos_y1=s_pos_y;
1118
1119         for (i=0; i < blockshape_x/BLOCK_SIZE; i++)
1120         {
1121             for (j=0; j < blockshape_y/BLOCK_SIZE;
j++)
1122             {
1123                 all_mv[block_x+i][block_y+j][
ref_frame][blocktype][0]=s_pos_x1;
1124                 all_mv[block_x+i][block_y+j][
ref_frame][blocktype][1]=s_pos_y1;
1125                 tmp_mv[0][pic_block_y+j][
pic_block_x+i+BLOCK_SIZE]=s_pos_x1;
1126                 tmp_mv[1][pic_block_y+j][
pic_block_x+i+BLOCK_SIZE]=s_pos_y1;
1127             }
1128         }
1129         best_inter_sad=current_inter_sad;
1130     }
1131 }

```

```

1132         tot_inter_sad += best_inter_sad;
1133     }
1134 }
1135
1136 /*
1137 Here we update which is the best inter mode. At the end
1138 we have the best inter mode.
1139 predframe_no: which reference frame to use for
1140 prediction
1141 img->multiframe_no: Index in the mref[] matrix used for
1142 prediction
1143 */
1144 if (tot_inter_sad <= min_inter_sad)
1145 {
1146     img->mb_mode=blocktype;
1147     img->b1c_size_h=inp_par->b1c_size[blocktype][0];
1148     img->b1c_size_v=inp_par->b1c_size[blocktype][1];
1149     predframe_no=ref_frame;
1150     img->multiframe_no=ref_inx;
1151     min_inter_sad=tot_inter_sad;
1152 }
1153 while (inp_par->b1c_size[++blocktype][0]==0 && blocktype<=
1154 7); /* only go through chosen blocksizes */
1155 }
1156
1157 /*
1158 tot_intra_sad is now the minimum SAD for intra. min_inter_sad is
1159 the best (min) SAD for inter (see above).
1160 Inter/intra is determined depending on which is smallest
1161 */
1162 if (tot_intra_sad < min_inter_sad)
1163 {
1164     img->mb_mode=img->imod+8*img->type; /* set intra mode in
1165 inter frame */
1166     for (hv=0; hv < 2; hv++)
1167         for (i=0; i < 4; i++)
1168             for (j=0; j < 4; j++)
1169                 tmp_mv[hv][img->block_y+j][img->block_x+i+4]=0;
1170 }
1171 else
1172 {
1173     img->mb_data[img->current_mb_nr].intraOrInter = INTER_MB;
1174     img->imod=INTRA_MB_INTER; /* Set inter mode */
1175     for (hv=0; hv < 2; hv++)
1176         for (i=0; i < 4; i++)
1177             for (j=0; j < 4; j++)
1178                 tmp_mv[hv][img->block_y+j][img->block_x+i+4]=
1179                     all_mv[i][j][predframe_no][img->mb_mode][hv];
1180 }
1181 return predframe_no;
1182 }
1183
1184 /*****
1185 **
1186 * Routine      find_sad
1187 *
1188 * Description: Do hadamard transform or normal SAD calculation.
1189 *              If Hadamard=1 4x4 Hadamard transform is performed and
1190 SAD of transform
1191 *              levels is calculated
1192 **
1193 */

```

```

1189  *   Input:          hadamard=0 : normal SAD
1190  *                   hadamard=1 : 4x4 Hadamard transform is performed and
                                SAD of transform
1191  *                               levels is calculated
1192  *
1193  *   Output:          SAD/hadamard transform value
1194  *
1195  *
1196  *****/
1197  int find_sad(struct img_par *img,int hadamard)
1198  {
1199      int i,j,m1[BLOCK_SIZE][BLOCK_SIZE],best_sad,current_sad;
1200
1201      current_sad=0;
1202      if (hadamard != 0)
1203      {
1204          best_sad=0;
1205          for (j=0; j < BLOCK_SIZE; j++)
1206          {
1207              m1[0][j]=img->m7[0][j]+img->m7[3][j];
1208              m1[1][j]=img->m7[1][j]+img->m7[2][j];
1209              m1[2][j]=img->m7[1][j]-img->m7[2][j];
1210              m1[3][j]=img->m7[0][j]-img->m7[3][j];
1211
1212              img->m7[0][j]=m1[0][j]+m1[1][j];
1213              img->m7[2][j]=m1[0][j]-m1[1][j];
1214              img->m7[1][j]=m1[2][j]+m1[3][j];
1215              img->m7[3][j]=m1[3][j]-m1[2][j];
1216          }
1217          for (i=0; i < BLOCK_SIZE; i++)
1218          {
1219              m1[i][0]=img->m7[i][0]+img->m7[i][3];
1220              m1[i][1]=img->m7[i][1]+img->m7[i][2];
1221              m1[i][2]=img->m7[i][1]-img->m7[i][2];
1222              m1[i][3]=img->m7[i][0]-img->m7[i][3];
1223
1224              img->m7[i][0]=m1[i][0]+m1[i][1];
1225              img->m7[i][1]=m1[i][0]-m1[i][1];
1226              img->m7[i][2]=m1[i][2]+m1[i][3];
1227              img->m7[i][3]=m1[i][3]-m1[i][2];
1228
1229              for (j=0; j < BLOCK_SIZE; j++)
1230                  best_sad += absm(img->m7[i][j]);
1231          }
1232          /*
1233          Calculation of normalized Hadamard transforms would require
          division by 4 here.
1234          However,we flevel that division by 2 is better (assuming we
          give the same penalty for
1235          bituse for Hadamard=0 and 1)
1236          */
1237          current_sad += best_sad/2;
1238      }
1239      else
1240      {
1241          for (i=0; i < BLOCK_SIZE; i++)
1242          {
1243              for (j=0; j < BLOCK_SIZE; j++)
1244              {
1245                  current_sad += absm(img->m7[i][j]);
1246              }
1247          }
1248      }
1249      return current_sad;
1250  }

```

```

1251
1252 /*****
1253  *
1254  *   Routine:      dct_luma
1255  *
1256  *   Description: The routine performs transform, quantization, inverse
1257  *                to the prediction and writes the result to the
1258  *                decoded luma frame. Includes the
1259  *                RD constrained quantization also.
1260  *
1261  *   Input:        block_x, block_y: Block position inside a macro block
1262  *                (0,4,8,12).
1263  *   Output:        nonzero: 0 if no levels are nonzero. 1 if there are
1264  *                nonzero levels.
1265  *                coeff_cost: Counter for nonzero coefficients, used to
1266  *                discard expensive levels.
1267  *****/
1268 int dct_luma(int block_x, int block_y, int *coeff_cost, struct img_par *
1269 img)
1270 {
1271     int sign(int a, int b);
1272
1273     int i, j, i1, j1, ilev, m5[4], m6[4], coeff_ctr, scan_loop_ctr;
1274     int qp_const, pos_x, pos_y, quant_set, level, scan_pos, run;
1275     int nonzero;
1276     int idx;
1277
1278     int scan_mode;
1279     int loop_rep;
1280     int coeff[16];
1281
1282     if (img->type == INTRA_IMG)
1283         qp_const = JQQ3; /* intra */
1284     else
1285         qp_const = JQQ4; /* inter */
1286
1287     pos_x = block_x / BLOCK_SIZE;
1288     pos_y = block_y / BLOCK_SIZE;
1289
1290     /* Horizontal transform */
1291
1292     for (j = 0; j < BLOCK_SIZE; j++)
1293     {
1294         for (i = 0; i < 2; i++)
1295         {
1296             i1 = 3 - i;
1297             m5[i] = img->m7[i][j] + img->m7[i1][j];
1298             m5[i1] = img->m7[i][j] - img->m7[i1][j];
1299         }
1300         img->m7[0][j] = (m5[0] + m5[1]) * 13;
1301         img->m7[2][j] = (m5[0] - m5[1]) * 13;
1302         img->m7[1][j] = m5[3] * 17 + m5[2] * 7;
1303         img->m7[3][j] = m5[3] * 7 - m5[2] * 17;
1304     }
1305
1306     /* Vertival transform */
1307
1308     for (i = 0; i < BLOCK_SIZE; i++)
1309     {

```

```

1309         for (j=0; j < 2; j++)
1310         {
1311             j1=3-j;
1312             m5[j]=img->m7[i][j]+img->m7[i][j1];
1313             m5[j1]=img->m7[i][j]-img->m7[i][j1];
1314         }
1315         img->m7[i][0]=(m5[0]+m5[1])*13;
1316         img->m7[i][2]=(m5[0]-m5[1])*13;
1317         img->m7[i][1]=m5[3]*17+m5[2]*7;
1318         img->m7[i][3]=m5[3]*7-m5[2]*17;
1319     }
1320
1321     /* Do prescan to decide one or two Q-levels for RD constrained
quantization*/
1322
1323     quant_set=img->qp;
1324     nonzero=FALSE;
1325
1326     if (img->imod == INTRA_MB_OLD && img->qp < 24)
1327     {
1328         scan_mode=DOUBLE_SCAN;
1329         loop_rep=2;
1330         idx=1;
1331     }
1332     else
1333     {
1334         scan_mode=SINGLE_SCAN;
1335         loop_rep=1;
1336         idx=0;
1337     }
1338
1339     for(scan_loop_ctr=0;scan_loop_ctr<loop_rep;scan_loop_ctr++) /* 2
times if double scan, 1 normal scan */
1340     {
1341         for (coeff_ctr=0;coeff_ctr < 16/loop_rep;coeff_ctr++) /*
8 times if double scan, 16 normal scan */
1342         {
1343             if (scan_mode==DOUBLE_SCAN)
1344             {
1345                 i=DBL_SCAN[coeff_ctr][0][scan_loop_ctr];
1346                 j=DBL_SCAN[coeff_ctr][1][scan_loop_ctr];
1347             }
1348             else
1349             {
1350                 i=SNGL_SCAN[coeff_ctr][0];
1351                 j=SNGL_SCAN[coeff_ctr][1];
1352             }
1353             coeff[coeff_ctr]=img->m7[i][j];
1354         }
1355         if (scan_mode==DOUBLE_SCAN)
1356             rd_quant(img,QUANT_LUMA_DBL,coeff);
1357         else
1358             rd_quant(img,QUANT_LUMA_SNG,coeff);
1359
1360         run=-1;
1361         scan_pos=scan_loop_ctr*9; /* for double scan; set first or
second scan position */
1362         for (coeff_ctr=0; coeff_ctr<16/loop_rep; coeff_ctr++)
1363         {
1364             if (scan_mode==DOUBLE_SCAN)
1365             {
1366                 i=DBL_SCAN[coeff_ctr][0][scan_loop_ctr];
1367                 j=DBL_SCAN[coeff_ctr][1][scan_loop_ctr];
1368             }
1369             else
1370

```

```

1371     {
1372         i=SNGL_SCAN[coeff_ctr][0];
1373         j=SNGL_SCAN[coeff_ctr][1];
1374     }
1375     run++;
1376     ilev=0;
1377
1378     level= absm(coeff[coeff_ctr]);
1379     if (level != 0)
1380     {
1381         nonzero=TRUE;
1382         if (level > 1)
1383             *coeff_cost += MAX_VALUE; /* set
1384             high cost, shall not be discarded */
1385         else
1386             *coeff_cost += COEFF_COST[run];
1387         img->cof[pos_x][pos_y][scan_pos][0][scan_mode]=sign(
1388             level,img->m7[i][j]);
1389         img->cof[pos_x][pos_y][scan_pos][1][scan_mode]=run;
1390         ++scan_pos;
1391         run=-1; /* reset zero level
1392         counter */
1393         ilev=level*JQ[quant_set][1];
1394     }
1395     img->m7[i][j]=sign(ilev,img->m7[i][j]);
1396 }
1397
1398 /* IDCT. */
1399 /* horizontal */
1400 for (j=0; j < BLOCK_SIZE; j++)
1401 {
1402     for (i=0; i < BLOCK_SIZE; i++)
1403     {
1404         m5[i]=img->m7[i][j];
1405     }
1406     m6[0]=(m5[0]+m5[2])*13;
1407     m6[1]=(m5[0]-m5[2])*13;
1408     m6[2]=m5[1]*7-m5[3]*17;
1409     m6[3]=m5[1]*17+m5[3]*7;
1410
1411     for (i=0; i < 2; i++)
1412     {
1413         i1=3-i;
1414         img->m7[i][j]=m6[i]+m6[i1];
1415         img->m7[i1][j]=m6[i]-m6[i1];
1416     }
1417 }
1418
1419 /* vertical */
1420
1421 for (i=0; i < BLOCK_SIZE; i++)
1422 {
1423     for (j=0; j < BLOCK_SIZE; j++)
1424     {
1425         m5[j]=img->m7[i][j];
1426     }
1427     m6[0]=(m5[0]+m5[2])*13;
1428     m6[1]=(m5[0]-m5[2])*13;
1429     m6[2]=m5[1]*7-m5[3]*17;
1430     m6[3]=m5[1]*17+m5[3]*7;
1431
1432     for (j=0; j < 2; j++)

```

```

1433     {
1434         j1=3-j;
1435         img->m7[i][j] =min(255,max(0,(m6[j]+m6[j1]+img->mpr[i+
block_x][j+block_y] *JQQ1+JQQ2)/JQQ1));
1436         img->m7[i][j1]=min(255,max(0,(m6[j]-m6[j1]+img->mpr[i+
block_x][j1+block_y] *JQQ1+JQQ2)/JQQ1));
1437     }
1438 }
1439
1440 /* Decoded block moved to frame memory */
1441
1442 for (j=0; j < BLOCK_SIZE; j++)
1443     for (i=0; i < BLOCK_SIZE; i++)
1444         imgY[img->pix_y+block_y+j][img->pix_x+block_x+i]=img->m7[i
][j];
1445
1446
1447     return nonzero;
1448 }
1449
1450 /*****
**
1451 *
1452 * Routine :    dct_chroma
1453 *
1454 * Description:
1455 *             Transform,quantization,inverse transform for chroma.
1456 *             The main reason why this is done in a separate
routine is the
1457 *             additional 2x2 transform of DC-coeffs. This routine
is called
1458 *             ones for each of the chroma components.
1459 *
1460 * Input:      uv      : Make difference between the U and V chroma
component
1461 *             cr_cbp: chroma coded block pattern
1462 *
1463 * Output:     cr_cbp: Updated chroma coded block pattern.
1464 *
1465 *
1466 *****/
1467 int dct_chroma(int uv,int cr_cbp,struct img_par *img)
1468 {
1469     int i,j,i1,j2,ilev,n2,n1,j1,mb_y,coeff_ctr,qp_const,pos_x,pos_y,
quant_set,level ,scan_pos,run;
1470     int m1[BLOCK_SIZE],m5[BLOCK_SIZE],m6[BLOCK_SIZE];
1471     int coeff[16];
1472
1473     if (img->type == INTRA_IMG)
1474         qp_const=JQQ3;
1475     else
1476         qp_const=JQQ4;
1477
1478     for (n2=0; n2 <= BLOCK_SIZE; n2 += BLOCK_SIZE)
1479     {
1480         for (n1=0; n1 <= BLOCK_SIZE; n1 += BLOCK_SIZE)
1481         {
1482
1483             /* Horizontal transform. */
1484             for (j=0; j < BLOCK_SIZE; j++)
1485             {
1486                 mb_y=n2+j;
1487                 for (i=0; i < 2; i++)
1488                 {
1489                     i1=3-i;

```

```

1490         m5[i]=img->m7[i+n1][mb_y]+img->m7[i1+n1][mb_y];
1491         m5[i1]=img->m7[i+n1][mb_y]-img->m7[i1+n1][mb_y];
1492     }
1493     img->m7[n1][mb_y]=(m5[0]+m5[1])*13;
1494     img->m7[n1+2][mb_y]=(m5[0]-m5[1])*13;
1495     img->m7[n1+1][mb_y]=m5[3]*17+m5[2]*7;
1496     img->m7[n1+3][mb_y]=m5[3]*7-m5[2]*17;
1497 }
1498
1499 /* Vertical transform. */
1500
1501 for (i=0; i < BLOCK_SIZE; i++)
1502 {
1503     j1=n1+i;
1504     for (j=0; j < 2; j++)
1505     {
1506         j2=3-j;
1507         m5[j]=img->m7[j1][n2+j]+img->m7[j1][n2+j2];
1508         m5[j2]=img->m7[j1][n2+j]-img->m7[j1][n2+j2];
1509     }
1510     img->m7[j1][n2+0]=(m5[0]+m5[1])*13;
1511     img->m7[j1][n2+2]=(m5[0]-m5[1])*13;
1512     img->m7[j1][n2+1]=m5[3]*17+m5[2]*7;
1513     img->m7[j1][n2+3]=m5[3]*7-m5[2]*17;
1514 }
1515 }
1516 }
1517
1518 /* 2X2 transform of DC coeffs. */
1519 m1[0]=(img->m7[0][0]+img->m7[4][0]+img->m7[0][4]+img->m7[4][4])/2;
1520 m1[1]=(img->m7[0][0]-img->m7[4][0]+img->m7[0][4]-img->m7[4][4])/2;
1521 m1[2]=(img->m7[0][0]+img->m7[4][0]-img->m7[0][4]-img->m7[4][4])/2;
1522 m1[3]=(img->m7[0][0]-img->m7[4][0]-img->m7[0][4]+img->m7[4][4])/2;
1523
1524 /* Quant of chroma 2X2 coeffs.*/
1525 quant_set=QP_SCALE_CR[img->qp];
1526 run=-1;
1527 scan_pos=0;
1528
1529 for (coeff_ctr=0; coeff_ctr < 4; coeff_ctr++)
1530     coeff[coeff_ctr]=m1[coeff_ctr];
1531
1532 rd_quant(img,QUANT_CHROMA_DC,coeff);
1533
1534 for (coeff_ctr=0; coeff_ctr < 4; coeff_ctr++)
1535 {
1536     run++;
1537     ilev=0;
1538     level =0;
1539
1540     level =(absm(coeff[coeff_ctr]));
1541     if (level != 0)
1542     {
1543         cr_cbp=max(1,cr_cbp);
1544         img->cofu[scan_pos][0][uv]=sign(level ,m1[coeff_ctr]);
1545         img->cofu[scan_pos][1][uv]=run;
1546         scan_pos++;
1547         run=-1;
1548         ilev=level*JQ[quant_set][1];
1549     }
1550     m1[coeff_ctr]=sign(ilev,m1[coeff_ctr]);
1551 }
1552 img->cofu[scan_pos][0][uv]=0;
1553
1554 /* Invers transform of 2x2 DC levels */
1555

```

```

15556 img->m7[0][0]=(m1[0]+m1[1]+m1[2]+m1[3])/2;
15557 img->m7[4][0]=(m1[0]-m1[1]+m1[2]-m1[3])/2;
15558 img->m7[0][4]=(m1[0]+m1[1]-m1[2]-m1[3])/2;
15559 img->m7[4][4]=(m1[0]-m1[1]-m1[2]+m1[3])/2;
1560
1561 /* Quant of chroma AC-coeffs. */
1562
1563 for (n2=0; n2 <= BLOCK_SIZE; n2 += BLOCK_SIZE)
1564 {
1565     for (n1=0; n1 <= BLOCK_SIZE; n1 += BLOCK_SIZE)
1566     {
1567         pos_x=n1/BLOCK_SIZE + 2*uv;
1568         pos_y=n2/BLOCK_SIZE + BLOCK_SIZE;
1569         run=-1;
1570         scan_pos=0;
1571
1572         for (coeff_ctr=1; coeff_ctr < 16; coeff_ctr++)
1573         {
1574             i=SNGL_SCAN[coeff_ctr][0];
1575             j=SNGL_SCAN[coeff_ctr][1];
1576             coeff[coeff_ctr-1]=img->m7[n1+i][n2+j];
1577         }
1578         rd_quant(img,QUANT_CHROMA_AC,coeff);
1579
1580         for (coeff_ctr=1; coeff_ctr < 16; coeff_ctr++)
1581         {
1582             i=SNGL_SCAN[coeff_ctr][0];
1583             j=SNGL_SCAN[coeff_ctr][1];
1584             ++run;
1585             ilev=0;
1586
1587             level=absm(coeff[coeff_ctr-1]);
1588             if (level != 0)
1589             {
1590                 cr_cbp=2;
1591                 img->cof[pos_x][pos_y][scan_pos][0][0]=sign(level,
1592 img->m7[n1+i][n2+j]);
1593                 img->cof[pos_x][pos_y][scan_pos][1][0]=run;
1594                 ++scan_pos;
1595                 run=-1;
1596                 ilev=level*JQ[quant_set][1];
1597             }
1598             img->m7[n1+i][n2+j]=sign(ilev,img->m7[n1+i][n2+j]);
1599             /* for use in IDCT */
1600         }
1601         img->cof[pos_x][pos_y][scan_pos][0][0]=0; /* EOB */
1602
1603         /* IDCT. */
1604
1605         /* Horizontal. */
1606         for (j=0; j < BLOCK_SIZE; j++)
1607         {
1608             for (i=0; i < BLOCK_SIZE; i++)
1609             {
1610                 m5[i]=img->m7[n1+i][n2+j];
1611             }
1612             m6[0]=(m5[0]+m5[2])*13;
1613             m6[1]=(m5[0]-m5[2])*13;
1614             m6[2]=m5[1]*7-m5[3]*17;
1615             m6[3]=m5[1]*17+m5[3]*7;
1616
1617             for (i=0; i < 2; i++)
1618             {
1619                 i1=3-i;
1620                 img->m7[n1+i][n2+j]=m6[i]+m6[i1];

```

```

1620         img->m7[n1+i1][n2+j]=m6[i]-m6[i1];
1621     }
1622 }
1623
1624 /*      Vertical. */
1625 for (i=0; i < BLOCK_SIZE; i++)
1626 {
1627     for (j=0; j < BLOCK_SIZE; j++)
1628     {
1629         m5[j]=img->m7[n1+i][n2+j];
1630     }
1631     m6[0]=(m5[0]+m5[2])*13;
1632     m6[1]=(m5[0]-m5[2])*13;
1633     m6[2]=m5[1]*7-m5[3]*17;
1634     m6[3]=m5[1]*17+m5[3]*7;
1635
1636     for (j=0; j < 2; j++)
1637     {
1638         j2=3-j;
1639         img->m7[n1+i][n2+j]=min(255,max(0,(m6[j]+m6[j2]+
1640         img->mpr[n1+i][n2+j]*JQQ1+JQQ2)/JQQ1));
1641         img->m7[n1+i][n2+j2]=min(255,max(0,(m6[j]-m6[j2]+
1642         img->mpr[n1+i][n2+j2]*JQQ1+JQQ2)/JQQ1));
1643     }
1644 }
1645
1646 /* Decoded block moved to memory */
1647 for (j=0; j < BLOCK_SIZE*2; j++)
1648     for (i=0; i < BLOCK_SIZE*2; i++)
1649     {
1650         imgUV[uv][img->pix_c_y+j][img->pix_c_x+i]= img->m7[i][j];
1651     }
1652
1653 return cr_cbp;
1654 }
1655
1656 void rd_quant(struct img_par *img, int scan_type,int *coeff)
1657 {
1658     int idx,coeff_ctr;
1659     int qp_const,intra_add;
1660
1661     int dbl_coeff_ctr;
1662     int level0,level1;
1663     int snr0;
1664     int dbl_coeff,k,k1,k2,rd_best,best_coeff_comb,rd_curr,snr1;
1665     int k0;
1666     int quant_set;
1667     int ilev,run,level;
1668     int no_coeff;
1669
1670     if (img->type == INTRA_IMG)
1671     {
1672         qp_const=JQQ3;
1673         intra_add=2;
1674     }
1675     else
1676     {
1677         qp_const=JQQ4;
1678         intra_add=0;
1679     }
1680     quant_set=img->qp;
1681
1682     switch (scan_type)
1683     {

```

```

1684     case QUANT_LUMA_SNG:
1685         quant_set=img->qp;
1686         idx=2;
1687         no_coeff=16;
1688         break;
1689     case QUANT_LUMA_AC:
1690         quant_set=img->qp;
1691         idx=2;
1692         no_coeff=15;
1693         break;
1694     case QUANT_LUMA_DBL:
1695         quant_set=img->qp;
1696         idx=1;
1697         no_coeff=8;
1698         break;
1699     case QUANT_CHROMA_DC:
1700         quant_set=QP_SCALE_CR[img->qp];
1701         idx=0;
1702         no_coeff=4;
1703         break;
1704     case QUANT_CHROMA_AC:
1705         quant_set=QP_SCALE_CR[img->qp];
1706         idx=2;
1707         no_coeff=15;
1708         break;
1709     default:
1710         break;
1711 }
1712
1713 dbl_coeff_ctr=0;
1714 for (coeff_ctr=0;coeff_ctr < no_coeff ;coeff_ctr++)
1715 {
1716     k0=coeff[coeff_ctr];
1717     k1=abs(k0);
1718
1719     if (dbl_coeff_ctr < MAX_TWO_LEVEL_COEFF) /* limit the number
1720 of 'twin' levels */
1721     {
1722         level0 = (k0*JQ[quant_set][0])/J20;
1723         level1 = (k1*JQ[quant_set][0]+JQ4)/J20; /* make positive
1724 summation */
1725         level1 = sign(level1,k0);/* set back sign on level */
1726     }
1727     else
1728     {
1729         level0 = (k1*JQ[quant_set][0]+qp_const)/J20;
1730         level0 = sign(level0,k0);
1731         level1 = level0;
1732     }
1733
1734     if (level0 != level1)
1735     {
1736         dbl_coeff = TRUE; /* decision is still open */
1737         dbl_coeff_ctr++; /* count number of coefficients
1738 with 2 possible levels */
1739     }
1740     else
1741         dbl_coeff = FALSE; /* level is decided */
1742
1743     snr0 = (12+intra_add)*level0*(64*level0 - (JQ[quant_set][0]*
1744 coeff[coeff_ctr])/J13); /* find SNR improvement */
1745
1746     level_arr[coeff_ctr][MTLC_POW]=0; /* indicates that all
1747 coefficients are decided */
1748

```

```

1745     for (k=0; k< MTLC_POW; k++)
1746     {
1747         level_arr[coeff_ctr][k]=level0;
1748         snr_arr[coeff_ctr][k]=snr0;
1749     }
1750     if (dbl_coeff)
1751     {
1752         snr1 = (12+intra_add)*level1*(64*level1 - (JQ[quant_set][0]
1753             ]*coeff[coeff_ctr])/J13);
1754         ilev= (int)pow(2,dbl_coeff_ctr-1);
1755         for (k1=ilev; k1<MTLC_POW; k1+=ilev*2)
1756         {
1757             for (k2=k1; k2<k1+ilev; k2++)
1758             {
1759                 level_arr[coeff_ctr][k2]=level1;
1760                 snr_arr[coeff_ctr][k2]=snr1;
1761             }
1762         }
1763     }
1764
1765     rd_best=0;
1766     best_coeff_comb= MTLC_POW; /* initial setting, used if no
1767     double decision coefficients */
1768     for (k=0; k < pow(2,dbl_coeff_ctr);k++) /* go through all
1769     combinations */
1770     {
1771         rd_curr=0;
1772         run=-1;
1773         for (coeff_ctr=0;coeff_ctr < no_coeff;coeff_ctr++)
1774         {
1775             run++;
1776             level=min(16,absm(level_arr[coeff_ctr][k]));
1777             if (level != 0)
1778             {
1779                 rd_curr += 64*COEFF_BIT_COST[idx][run][level-1]+
1780                 snr_arr[coeff_ctr][k];
1781                 run = -1;
1782             }
1783             if (rd_curr < rd_best)
1784             {
1785                 rd_best=rd_curr;
1786                 best_coeff_comb=k;
1787             }
1788         }
1789         coeff[coeff_ctr]=level_arr[coeff_ctr][best_coeff_comb];
1790     }
1791     return;
1792 }
1793
1794 /* control the sign of a with b */
1795 int sign(int a,int b)
1796 {
1797     int x;
1798     x=absm(a);
1799     if (b >= 0)
1800         return x;
1801     else
1802         return -x;
1803 }
1804
1805
1806

```

```

1  /*****
2  **
3  *   global.h for H.26L encoder.
4  *   Copyright (C) 1999   Telenor Satellite Services,Norway
5  *                           Ericsson Radio Systems, Sweden
6  *
7  *   Contacts:
8  *   Inge Lille-Langøy          <inge.lille-langoy@telenor.com>
9  *
10 *   Telenor Satellite Services
11 *   Keyzers gt.13              tel.:   +47 23 13 86 98
12 *   N-0130 Oslo,Norway        fax.:   +47 22 77 79 80
13 *
14 *   Rickard Sjoberg            <rickard.sjoberg@era.ericsson.se>
15 *
16 *   Ericsson Radio Systems
17 *   KI/ERA/T/VV
18 *   164 80 Stockholm, Sweden
19 *
20 *****/
21 #ifndef _GLOBAL_H_
22 #define _GLOBAL_H_
23
24 #include <stdio.h>
25 #include "defines.h"
26
27
28 /*****
29 *
30 *   T y p e       d e f i n i t i o n s       f o r       T M L
31 *
32 *****/
33
34 typedef unsigned char byte;
35
36 /* Introduced by TOM */
37 /* Data Partitioning Modes */
38 typedef enum
39 {
40     PAR_DP_1,      /* no data partitioning is supported */
41     PAR_DP_2,      /* data partitioning with 2 partitions */
42     PAR_DP_4       /* data partitioning with 4 partitions */
43 } PAR_DP_TYPE;
44
45 /* Output File Types */
46 typedef enum
47 {
48     PAR_OF_26L,    /* Current TML description */
49     PAR_OF_SLICE,  /* Slice File Format */
50     PAR_OF_NAL     /* Current NAL Format */
51 } PAR_OF_TYPE;
52 /* End Introduced by TOM */
53
54
55 typedef enum {
56     FALSE,
57     TRUE
58 } Boolean;
59
60 typedef enum {
61     SE_HEADER,

```

```

62     SE_PTYPE,
63     SE_MBTYPE,
64     SE_REFFRAME,
65     SE_INTRAPREDMODE,
66     SE_MVD,
67     SE_CBP_INTRA,
68     SE_LUM_DC_INTRA,
69     SE_CHR_DC_INTRA,
70     SE_LUM_AC_INTRA,
71     SE_CHR_AC_INTRA,
72     SE_CBP_INTER,
73     SE_LUM_DC_INTER,
74     SE_CHR_DC_INTER,
75     SE_LUM_AC_INTER,
76     SE_CHR_AC_INTER,
77     SE_BFRAME,
78     SE_EOS,
79     SE_MAX_ELEMENTS /* number of maximum syntax elements */
80 } SE_type;          /* substituting the definitions in element.h */
81
82
83 typedef enum {
84     INTER_MB,
85     INTRA_MB_4x4,
86     INTRA_MB_16x16
87 } IntraInterDecision;
88
89 typedef enum {
90     BITS_TOTAL_MB,
91     BITS_HEADER_MB,
92     BITS_INTER_MB,
93     BITS_CBP_MB,
94     BITS_COEFF_Y_MB,
95     BITS_COEFF_UV_MB,
96     MAX_BITCOUNTER_MB
97 } BitCountType;
98
99 typedef enum {
100     SINGLE_SCAN,
101     DOUBLE_SCAN
102 } ScanMode;
103
104 typedef enum {
105     NO_SLICES,
106     FIXED_MB,
107     FIXED_RATE,
108     CALLBACK
109 } SliceMode;
110
111 typedef enum {
112     UVLC,
113     CABAC
114 } SymbolMode;
115
116
117 /*****
118  *
119  * Data types for CABAC
120  *
121  */
122
123 typedef struct
124 {

```

```

125     unsigned int    Elow, Ehigh;
126     unsigned int    Ebuffer;
127     unsigned int    Ebits_to_go;
128     unsigned int    Ebits_to_follow;
129     byte             *Ecodestrm;
130     int              *Ecodestrm_len;
131 } EncodingEnvironment; /* struct to characterize
the state of the arithmetic coding engine */
132
133 typedef EncodingEnvironment *EncodingEnvironmentPtr;
134
135
136 typedef struct
137 {
138     unsigned int    cum_freq[2]; /* cumulated frequency
counts */
139     Boolean         in_use; /* flag for context in
use */
140     unsigned int    max_cum_freq; /* maximum
frequency count */
141
142 } BiContextType; /* struct for context
management */
143
144 typedef BiContextType *BiContextTypePtr;
145
146
147 /*****
148  * C O N T E X T S   F O R   T M L   S Y N T A X   E L E M E N T S
149  *****/
150 */
151
152 #define NUM_MB_TYPE_CTX 10
153 #define NUM_MV_RES_CTX 10
154 #define NUM_REF_NO_CTX 6
155
156 typedef struct
157 {
158     BiContextTypePtr mb_type_contexts[2];
159     BiContextTypePtr mv_res_contexts[2];
160     BiContextTypePtr ref_no_contexts;
161 } MotionInfoContexts;
162
163 #define NUM_IPR_CTX 2
164 #define NUM_CBP_CTX 4
165 #define NUM_TRANS_TYPE 9
166 #define NUM_LEVEL_CTX 4
167 #define NUM_RUN_CTX 2
168
169 typedef struct
170 {
171     BiContextTypePtr ipr_contexts[6];
172     BiContextTypePtr cbp_contexts[2][3];
173     BiContextTypePtr level_context[NUM_TRANS_TYPE];
174     BiContextTypePtr run_context[NUM_TRANS_TYPE];
175 } TextureInfoContexts;
176
177 /***** end of data type definition for CABAC
*****/
178
179
180 /*****
181  * N e w   D a t a   t y p e s   f o r   T M L
182  *****/

```

```

183  */
184
185  typedef struct syntaxelement {
186      int                type;                /* type of syntax
187      element for data part.*/
188      int                value1;              /* numerical value of
189      syntax element */
190      int                value2;              /* for blocked
191      symbols, e.g. run/level */
192      int                len;                 /* length of
193      code */
194      int                inf;                 /* info part of
195      UVLC code */
196      unsigned int        bitpattern; /* UVLC
197      bitpattern */
198      int                context;             /* CABAC
199      context */
200      #if TRACE
201      char                tracestring[100];   /* trace
202      string */
203      #endif
204
205      void                (*mapping)(int value1, int value2, int* len_ptr,
206      int* info_ptr); // for mapping of syntaxElement to UVLC
207      void                (*writing)(struct syntaxelement *, struct inp_par *,
208      struct img_par *, EncodingEnvironmentPtr);
209
210      /* used for
211      CABAC: refers to actual coding method of each
212      individual
213      syntax element type */
214  } SyntaxElement;
215
216  typedef struct macroblock
217  {
218      int                currSEnr;            /* number
219      of current syntax element */
220      int                slice_nr;            /* slice
221      number to which the MB belongs */
222      int                qp;                 /* for
223      future use */
224      int                intraOrInter;
225      int                bitcounter[MAX_BITCOUNTER_MB];
226      struct macroblock  *mb_available[3][3]; /*
227      pointer to neighboring MBs in a 3x3 window of current MB, which
228      is located at [1][1] */
229
230      /* NULL pointer identifies neighboring MBs which
231      are unavailable */
232
233      /* some storage of macroblock syntax elements for global access */
234      int                mb_type;
235      int                mb_imode;
236      int                ref_frame;
237      int                mvd[2][BLOCK_MULTIPLE][
238      BLOCK_MULTIPLE][2]; /* indices correspond to
239      [forw,backw][block_y][block_x][x,y] */
240      int                intra_pred_modes[
241      BLOCK_MULTIPLE*BLOCK_MULTIPLE];
242      int                cbp;
243
244  } Macroblock;
245
246  typedef struct {

```

```

224     int                byte_pos;                // current
           position in bitstream;
225     int                bits_to_go;              // current bitcounter
226     byte               byte_buf;                // current buffer
           for last written byte
227     int                stored_byte_pos;         // storage for position
           in bitstream;
228     int                stored_bits_to_go;       // storage for bitcounter
229     byte               stored_byte_buf;        // storage for buffer of
           last written byte
230
231     byte               *streamBuffer;           // actual buffer for
           written bytes
232
233 } Bitstream;
234
235 typedef struct datapartition {
236
237     Bitstream           *bitstream;
238     EncodingEnvironment ee_cabac;
239
240     int                 (*writeSyntaxElement)(
           SyntaxElement *, struct img_par *, struct inp_par *, struct
           datapartition *);
241
           //
242     virtual function;           // actual
           method depends on chosen data partition and
243
           //
           entropy coding method
244 } DataPartition;
245
246 typedef struct {
247     int                 picture_id;
248     int                 slice_nr;               // not necessary but o.k.
249     int                 qp;
250     int                 format;
251     int                 picture_type;           // picture type
252     int                 start_mb_nr;
253     int                 dp_mode;               // data partitioning mode
254     int                 max_part_nr;           // number of different
           partitions
255     int                 eos_flag;              // end of sequence flag
256     DataPartition       *partArr;              // array of partitions
257     MotionInfoContexts  *mot_ctx;              // pointer to struct of
           context models for use in CABAC
258     TextureInfoContexts *tex_ctx;              // pointer to struct of
           context models for use in CABAC
259
260
261     Boolean             (*slice_too_big)(int bits_slice); //
           for use of callback functions
262
263 } Slice;
264
           /***** ~DM *****/
265
266 /* GH: old fixed mem allocation for picture sizes up to CIF*/
267 /*int  refFrArr[72][88];*/           /* Array for reference frames
           of each block */
268 /*byte imgY[288][352]; */           /* Encoded luma images */
269 /*byte imgY_org[288][352]; */       /* Reference luma image */
270 /*byte imgY_pf[288][352]; */       /* Post filter luma image */
271 /*byte imgUV[2][144][176]; */       /* Encoded chroma images */
272 /*byte imgUV_org[2][144][176]; */   /* Reference chroma images */
273 /*byte imgUV_pf[2][144][176]; */    /* Post filter chroma images */

```

```

274  /* fix from ver 4.1 */
275  /*byte loopb[91][75]; */          /* array containing filter
strenght for 4x4 luma block */
276  /*byte loopc[47][39]; */          /* array containing filter
strenght for 4x4 chroma block */
277  /*byte mref[MAX_MULT_PRED][1152][1408]; */ /* 1/4 pix luma */
278  /*byte mcef[MAX_MULT_PRED][2][352][288]; */ /* pix chroma */
279  /*byte mref_P[1152][1408];      1/4 pix luma for next P picture*/
280  /*byte mcef_P[2][352][288];     pix chroma for next P picture*/
281  /*int  imgY_tmp[576][704];*/
282
283  /* GH: global picture format dependend buffers, mem allocation in
image.c*/
284  byte  **imgY;          /* Encoded luma images */
285  byte  ***imgUV;        /* Encoded cromia images */
286  byte  **imgY_org;      /* Reference luma image */
287  byte  ***imgUV_org;    /* Reference cromia image */
288  byte  **imgY_pf;       /* Post filter luma image */
289  byte  ***imgUV_pf;     /* Post filter cromia image */
290  byte  **loopb;         /* array containing filter strenght for
4x4 luma block */
291  byte  **loopc;         /* array containing filter strenght for
4x4 chroma block */
292  byte  ***mref;         /* 1/4 pix luma */
293  byte  *****mcef;     /* pix chroma */
294  int    **img4Y_tmp;     /* for quarter pel interpolation */
295  byte  **imgY_tmp;       /* for loop filter */
296  byte  ***imgUV_tmp;
297  int    ***tmp_mv;       /* motion vector buffer */
298  int    **refFrArr;     /* Array for reference frames of each
block */
299
300  /* B pictures */
301  // motion vector : forward, backward, direct
302  int    ***tmp_fwMV;     //[2][72][92];
303  int    ***tmp_bwMV;     //[2][72][92];
304  int    ***dfMV;         //[2][72][92];
305  int    ***dbMV;         //[2][72][92];
306  int    **fw_refFrArr;   //[72][88];
307  int    **bw_refFrArr;   //[72][88];
308  byte  **mref_P;         //for B-frames: 1/4 pix luma for next P picture
309  byte  ***mcef_P;        //for B-frames: pix chroma for next P picture
310  byte  **nextP_imgY;
311  byte  ***nextP_imgUV;
312
313  int    Bframe_ctr, frame_no, nextP_tr;
314  int    tot_time;
315
316  char  errortext[300];
317
318  struct snr_par
319  {
320      float snr_y;        /* current Y SNR
*/
321      float snr_u;        /* current U SNR
*/
322      float snr_v;        /* current V SNR
*/
323      float snr_y1;       /* SNR Y(dB) first frame
*/
324      float snr_u1;       /* SNR U(dB) first frame
*/
325      float snr_v1;       /* SNR V(dB) first frame
*/
326      float snr_ya;       /* Average SNR Y(dB) remaining frames
*/

```

```

327     float snr_ua;                /* Average SNR U(dB) remaining frames
    */
328     float snr_va;                /* Average SNR V(dB) remaining frames
    */
329 };
330
331 struct inp_par                    /* all input parameters */
332 {
333     int no_frames;                /* number of frames to be
    encoded */
334     int qp0;                      /* QP of first
    frame */
335     int qpN;                      /* QP of remaining
    frames */
336     int jumpd;                   /* number of frames to skip in input
    sequence (e.g 2 takes frame 0,3,6,9...) */
337     int hadamard;                /* 0: 'normal' SAD in 1/3 pixel
    search. 1: use 4x4 Haphazard transform and ' */
338                                /* Sum of absolute transform
    difference' in 1/3 pixel
    search */
339     int search_range;            /* search range - integer pel search
    and 16x16 blocks. The search window is */
340                                /* generally around the predicted
    vector. Max vector is 2xmcrange. For
    8x8 */
341                                /* and 4x4 block sizes the search
    range is 1/2 of that for 16x16
    blocks. */
342     int no_mulptred;            /* 1: prediction from the last frame
    only. 2: prediction from the last or */
343                                /* second last frame etc. Maximum 5
    frames
    */
344     int img_format;              /* GH: 0=SUB_QCIF, 1=QCIF; 2=CIF,
    3=CIF_4, 4=CIF_16, 5=CUSTOM */
345     int img_width;               /* GH: if CUSTOM image format is
    chosen, use this size */
346     int img_height;              /* GH: width and height must be a
    multiple of 16 pels */
347     int yuv_format;              /* GH: YUV format (0=4:0:0, 1=4:2:0,
    2=4:2:2, 3=4:4:4, currently only 4:2:0 is supported) */
348     int color_depth;             /* GH: YUV color depth per component
    in bit/pel (currently only 8 bit/pel is supported) */
349     int intra_upd;               /* For error robustness. 0: no
    special action. 1: One GOB/frame is intra coded */
350                                /* as regular 'update'. 2: One GOB
    every 2 frames is intra coded
    etc. */
351                                /* In connection with this intra
    update, restrictions is put on motion
    vectors */
352                                /* to prevent errors to propagate
    from the past */
353     int write_dec;               /* write decoded image to file, for
    debugging */
354     int blc_size[8][2];          /* array for different block sizes */
355     int slice_mode;              /* Indicate what algorithm to use for
    setting slices */
356     int slice_argument;          /* Argument to the specified slice
    algorithm */
357     char infile[100];            /* YUV 4:2:0 input format */
358     char outfile[100];           /* H.26L compressed output bitstream */
359
360     /* B pictures */
361     char dec_file[100], dec_file2[100];

```

```

362     int successive_Bframe;           /* number of B frames that will be
        used */
363     int qpB;                         /* QP of B frames */
364
365     /* Introduced by TOM */
366     int symbol_mode;                 /* Specifies the mode the symbols are
        mapped on bits */
367     int of_mode;                     /* Specifies the mode of the output
        file */
368     int partition_mode;              /* Specifies the mode of data
        partitioning */
369 };
370
371 struct img_par
372 {
373     int number;                      /* current image number to be encoded*/
374     int current_mb_nr;
375     int current_slice_nr;
376     int type;
377     int no_multpred;                 /* 1: prediction from the last frame
        only. 2: prediction from the last or */
378                                     /* second last frame etc. Maximum
        MAX_MULT_PRED
        frames */
379     int multframe_no;
380     int qp;                          /* quant for the current frame */
381     int frame_cycle;
382     int framerate;
383     int width;                       /* Number of pels */
384     int width_cr;                    /* Number of pels chroma */
385     int height;                      /* Number of lines */
386     int height_cr;                   /* Number of lines chroma */
387     int height_err;                 /* For y vector limitation for error
        robustness*/
388     int mb_y;                        /* current MB vertical */
389     int mb_x;                        /* current MB horizontal */
390     int block_y;                     /* current block vertical */
391     int block_x;                     /* current block horizontal */
392     int subblock_y;                  /* DM: current subblock vertical */
393     int subblock_x;                  /* DM: current subblock horizontal */
394     int pix_y;                       /* current pixel vertical */
395     int pix_x;                       /* current pixel horizontal */
396     int mb_y_upd;
397     int mb_y_intra;                  /* which GOB to intra code */
398     int mb_mode;                     /* MB mode relevant for inter images,
        for intra images are all MB intra*/
399     int imod;                        /* new/old intra modes + inter */
400     int pix_c_y;                     /* current pixel chroma vertical */
401     int block_c_x;                   /* current block chroma vertical */
402     int pix_c_x;                     /* current pixel chroma horizontal */
403     int blc_size_h;                  /* block size for motion search,
        horizontal */
404     int blc_size_v;                  /* block size for motion search,
        vertical */
405     int kac;                         /* any AC coeffs */
406     int spiral_search[2][6561];      /* To obtain spiral search */
407     int **ipredmode;                 /* GH ipredmode[90][74];prediction
        mode for inter frames */ /* fix from ver 4.1 */
408
409     /* some temporal buffers */
410     int mpr[6][4][4];                /* all 5 prediction modes */
411     int mpr_2[5][16][16];            /* all 4 new intra prediction modes */
412     int mv[4][4][MAX_MULT_PRED][9][2]; /* motion vectors for all
        block types and all reference frames */
413     int mpr[16][16];                 /* current best prediction mode */
414     int m7[16][16];                 /* the diff pixel values between

```

```

original image and prediction */
415 int cof[4][6][18][2][2]; /* coefficients */
416 int cofu[5][2][2]; /* coefficients chroma */
417 /* DM: 030501 */
418 byte tmp_loop_Y[BLOCK_MULTIPLE+2][BLOCK_MULTIPLE+2];
/* temporal storage for LUMA loop-filter strength */
419 byte tmp_loop_UV[BLOCK_MULTIPLE/2+2][BLOCK_MULTIPLE/2+2]; /*
temporal storage for CHROMA loop-filter strength */
420 /* ~DM */
421
422 /* DM: 030501 */
423 Slice *currentSlice; /* pointer to current
Slice data struct */
424 Macroblock *mb_data; /* array containing all MBs
of a whole frame */
425 SyntaxElement MB_SyntaxElements[MAX_SYMBOLS_PER_MB]; /*
temporal storage for all chosen syntax elements of one MB */
426 /* ~DM */
427
428 int quad[256]; /* Array containing square
values, used for snr computation */
/* Values are limited to
5000 for pixel differences over 70 (sqr(5000)). */
429 int mv_bituse[512];
430 int *slice_numbers;
431
432 byte li[8]; /* 8 pix input for new filter routine
loop() */
433 byte lu[6]; /* 4 pix output for new filter routine
loop() */
434
435 int lpfilter[6][602]; /* for use in loopfilter */
436
437 /* B pictures */
438 int tr;
439 int b_interval;
440 int p_interval;
441 int b_frame_to_code;
442 int fw_mb_mode;
443 int fw_multiframe_no;
444 int fw_blc_size_h;
445 int fw_blc_size_v;
446 int bw_mb_mode;
447 int bw_multiframe_no;
448 int bw_blc_size_h;
449 int bw_blc_size_v;
450 int p_fwMV[4][4][MAX_MULT_PRED][9][2]; // for MVDFW
451 int p_bwMV[4][4][MAX_MULT_PRED][9][2]; // for MVDBW
452 int blk_bituse[10]; // it is included when getting bid_sad
453 };
454
455
456 struct stat_par /* statistics */
457 {
458 int quant0; /* quant for the first frame */
459 int quant1; /* average quant for the remaining
frames */
460 float bitr; /* bit rate for current frame, used
only for output til terminal*/
461 float bitr0; /* stored bit rate for the first
frame*/
462 float bitrate; /* average bit rate for the sequence
except first frame*/
463 int bit_ctr; /* counter for bit usage */
464 int bit_ctr_0; /* stored bit use for the first frame*/
465 int bit_ctr_n; /* bit usage for the current frame*/

```

```

466     int    bit_slice;                /* number of bits in current slice */
467     int    bit_use_mode_inter[33]; /* statistics of bit usage */
468     int    mode_use_intra[25];      /* Macroblock mode usage for Intra
frames */
469     int    mode_use_inter[33];
470
471     /* B pictures */
472     int    *mode_use_Bframe;
473     int    *bit_use_mode_Bframe;
474     int    bit_ctr_P;
475     int    bit_ctr_B;
476     float  bitrate_P;
477     float  bitrate_B;
478     int    bit_use_head_mode[3];
479     int    tmp_bit_use_cbp[3];
480     int    bit_use_coeffY[3];
481     int    bit_use_coeffC[3];
482 };
483
484 /* files */
485 FILE *p_dec;          /* internal decoded image for debugging*/
486 FILE *p_out;          /* H.26L output bitstream */
487 FILE *p_stat;         /* status file for the last encoding session */
488 FILE *p_log;          /* SNR file */
489 FILE *p_in;           /* YUV */
490 FILE *p_datpart;      /* file to write bitlength and id of all
partitions */
491
492 #if TRACE
493 FILE *p_trace;        /* Trace file */
494 #endif
495
496
497 /*****
*
498  * P r o t o t y p e s   f o r   T M L
499
*****
*
500 */
501
502
503 void intrapred_luma(struct img_par *img,int CurrPixX,int CurrPixY);
504 void init(struct img_par *img);
505 void find_snr(struct snr_par *snr,struct img_par *img);
506 void oneforthpix(struct img_par *img, struct inp_par *inp);
507 void oneforthpix_2(struct img_par *img);
508 int  encode_oneIorP_Frame(struct img_par *img,struct inp_par *inp,
struct stat_par *stat, struct snr_par *snr);
509 int  encode_oneB_Frame(struct img_par *img,struct inp_par *inp,struct
stat_par *stat, struct snr_par *snr);
510 int  find_sad(struct img_par *img,int hadamard);
511 int  dct_luma(int pos_mb1,int pos_mb2,int *cnt_nonz,struct img_par *
img);
512 int  dct_chroma(int uv,int i11,struct img_par *img);
513 int  motion_search(struct inp_par *inp,struct img_par *img, int isi);
514 void levrn_linfo_c2x2(int level,int run,int *len,int *info);
515 void levrn_linfo_intra(int level,int run,int *len,int *info);
516 void levrn_linfo_inter(int level,int run,int *len,int *info);
517 int  sign(int a,int b);
518 void intrapred_chroma(struct img_par *img,int,int,int uv);
519 void rd_quant(struct img_par *img, int no_coeff,int *coeff);
520 void intrapred_luma_2(struct img_par *img);
521 int  find_sad2(struct img_par *img,int *intra_mode);
522 void dct_luma2(struct img_par *img, int);
523

```

```

524 void init_conf(struct inp_par *inp, char *config_filename);
525 void init_img(struct inp_par *inp, struct img_par *img);
526 void init_stat(struct inp_par *inp, struct stat_par *stat);
527 void report(struct inp_par *inp, struct img_par *img, struct stat_par
    *stat, struct snr_par *snr);
528 void information_init(struct inp_par *inp, char *config_filename);
529 void init_frame(struct img_par *img, struct inp_par *inp, struct
    stat_par *stat);
530 void select_picture_type(struct img_par *img, struct inp_par *inp,
    SyntaxElement *symbol);
531 void read_one_new_frame(struct img_par *img, struct inp_par *inp);
532 void write_reconstructed_image(struct img_par *img, struct inp_par *
    inp);
533 void init_loop_filter(struct img_par *img);
534 int loop(struct img_par *img, int ibl, int ibr, int longFilt, int
    chroma);
535 void loopfilter(struct img_par *img);
536
537 /* GH: Added for dynamic mem allocation*/
538 int get_mem4global_buffers(struct inp_par *inp, struct img_par *img);
539 void free_mem4global_buffers(struct inp_par *inp, struct img_par *img
    );
540 int get_mem2D(byte ***array2D, int rows, int columns);
541 int get_mem2Dint(int ***array2D, int rows, int columns);
542 int get_mem3D(byte ****array2D, int frames, int rows, int columns);
543 int get_mem3Dint(int ****array3D, int frames, int rows, int columns);
544 void no_mem_exit(int code);
545 /* end GH*/
546
547
548 /* DM: Added for (re-) structuring the TML soft */
549 int encode_one_frame(struct img_par *img, struct inp_par *inp,
    struct stat_par *stat, struct snr_par *snr);
550 void encode_one_slice(struct img_par *img, struct inp_par *inp,
    struct stat_par *stat, SyntaxElement *se);
551 void malloc_slice(struct inp_par *inp, struct img_par *img);
552 void free_slice(struct inp_par *inp, struct img_par *img);
553 void init_slice(struct img_par *img, struct inp_par *inp);
554 void encode_one_macroblock(struct img_par *img, struct inp_par *inp,
    struct stat_par *stat);
555 void start_macroblock(struct img_par *img, struct inp_par *inp,
    struct stat_par *stat);
556
557 void terminate_macroblock(struct img_par *img, struct inp_par *inp,
    struct stat_par *stat,
558
    Boolean *
    end_of_slice,
    Boolean *
    recode_macroblo
    ck);
559 void write_one_macroblock(struct img_par *img, struct inp_par *inp,
    struct stat_par *stat);
560 void proceed2nextMacroblock(struct img_par *img, struct inp_par *inp
    , struct stat_par *stat);
561 void LumaResidualCoding_P(struct img_par *img, struct inp_par *inp);
562 void ChromaCoding_P(int *cr_cbp, struct img_par *img);
563 void SetRefFrameInfo_P(struct img_par *img);
564 void SetLoopfilterStrength_P(struct img_par *img);
565 int MakeIntraPrediction(struct img_par *img, struct inp_par *inp,
    int *intra_pred_mode_2);
566 void CheckAvailabilityOfNeighbors(struct img_par *img);
567 void writeMotionInfo2NAL_Pframe(struct img_par *img, struct inp_par
    *inp);
568 void writeCBPandCoeffs2NAL(struct img_par *img, struct inp_par *inp);
569 int writeStartcode2buffer(struct img_par *img, struct inp_par *inp
    , SyntaxElement *se);

```

```
570 void writeEOS2buffer(struct img_par *img, struct inp_par *inp);
571 void writeUVLC2buffer(SyntaxElement *se, struct img_par *img,
    struct inp_par *inp, Bitstream *currStream);
572 int writeSyntaxElement_UVLC(SyntaxElement *se, struct img_par *img
    , struct inp_par *inp, DataPartition *this_dataPart);
573 int symbol2uvlc(SyntaxElement *se);
574 void n_linf0(int n, int *len, int *info);
575 void n_linf02(int n, int dummy, int *len, int *info);
576 void intrapred_linf0(int ipred1, int ipred2, int *len, int *info);
577 void mvd_linf02(int mvd, int dummy, int *len, int *info);
578 void cbp_linf0_intra(int cbp, int dummy, int *len, int *info);
579 void cbp_linf0_inter(int cbp, int dummy, int *len, int *info);
580 #if TRACE
581 void trace2out(SyntaxElement *se);
582 #endif
583 Boolean dummy_slice_too_big(int bits_slice);
584 /* ~DM */
585
586 /* DM: Added for CABAC */
587 void arienco_start_encoding(EncodingEnvironmentPtr eep, unsigned char
    *code_buffer, int *code_len );
588 int arienco_bits_written(EncodingEnvironmentPtr eep);
589 void arienco_done_encoding(EncodingEnvironmentPtr eep);
590 void biari_init_context( BiContextTypePtr ctx, int ini_count_0, int
    ini_count_1, int max_cum_freq );
591 void biari_copy_context( BiContextTypePtr ctx_orig, BiContextTypePtr
    ctx_dest );
592 void biari_print_context( BiContextTypePtr ctx );
593 void rescale_cum_freq(BiContextTypePtr bi_ct);
594 void biari_encode_symbol(EncodingEnvironmentPtr eep, signed short
    symbol, BiContextTypePtr bi_ct );
595 MotionInfoContexts* create_contexts_MotionInfo(void);
596 TextureInfoContexts* create_contexts_TextureInfo(void);
597 void init_contexts_MotionInfo(struct img_par *img, MotionInfoContexts
    *enco_ctx, int ini_flag);
598 void init_contexts_TextureInfo(struct img_par *img,
    TextureInfoContexts *enco_ctx, int ini_flag);
599 void delete_contexts_MotionInfo(MotionInfoContexts *enco_ctx);
600 void delete_contexts_TextureInfo(TextureInfoContexts *enco_ctx);
601 void writeHeaderToBuffer(struct inp_par *inp, struct img_par *img);
602 void writeEOSToBuffer(struct inp_par *inp, struct img_par *img);
603 int writeSyntaxElement_CABAC(SyntaxElement *se, struct img_par *img,
    struct inp_par *inp, DataPartition *this_dataPart);
604 void writeMB_typeInfo2Buffer_CABAC(SyntaxElement *se, struct inp_par *
    inp, struct img_par *img, EncodingEnvironmentPtr eep_dp);
605 void writeIntraPredMode2Buffer_CABAC(SyntaxElement *se, struct inp_par
    *inp, struct img_par *img, EncodingEnvironmentPtr eep_dp);
606 void writeRefFrame2Buffer_CABAC(SyntaxElement *se, struct inp_par *inp
    , struct img_par *img, EncodingEnvironmentPtr eep_dp);
607 void writeMVD2Buffer_CABAC(SyntaxElement *se, struct inp_par *inp,
    struct img_par *img, EncodingEnvironmentPtr eep_dp);
608 void writeCBP2Buffer_CABAC(SyntaxElement *se, struct inp_par *inp,
    struct img_par *img, EncodingEnvironmentPtr eep_dp);
609 void writeRunLevel2Buffer_CABAC(SyntaxElement *se, struct inp_par *inp
    , struct img_par *img, EncodingEnvironmentPtr eep_dp);
610 void writeBiDirBlkSize2Buffer_CABAC(SyntaxElement *se, struct inp_par
    *inp, struct img_par *img, EncodingEnvironmentPtr eep_dp);
611 void writeBiMVD2Buffer_CABAC(SyntaxElement *se, struct inp_par *inp,
    struct img_par *img, EncodingEnvironmentPtr eep_dp);
612 /* ~DM */
613
614 /* Introduced by TOM */
615 void error(char *text);
616 int start_sequence(struct inp_par *inp);
617 int terminate_sequence(struct img_par *img, struct inp_par *inp);
618 int start_slice(struct img_par *img, struct inp_par *inp,
```

```
SyntaxElement *sym);
619 int terminate_slice(struct img_par *img, struct inp_par *inp, struct
    stat_par *stat);
620
621 /* B pictures */
622 void copy2mref(struct img_par *img);
623 int motion_search_Bframe(int tot_intra_sad, struct img_par *img,
    struct inp_par *inp);
624 int get_fwMV(int *min_fw_sad, struct img_par *img, struct inp_par *inp
    , int tot_intra_sad);
625 void get_bwMV(int *min_bw_sad, struct img_par *img, struct inp_par *
    inp_par);
626 void get_bid(int *bid_sad, int fw_predframe_no, struct img_par *img,
    struct inp_par *inp);
627 void get_dir(int *dir_sad, struct img_par *img, struct inp_par *inp);
628 void compare_sad(int tot_intra_sad, int fw_sad, int bw_sad, int
    bid_sad, int dir_sad, struct img_par *img);
629 void LumaResidualCoding_B(struct img_par *img);
630 void ChromaCoding_B(int *cr_cbp, struct img_par *img);
631 void SetLoopfilterStrength_B(struct img_par *img);
632 void SetRefFrameInfo_B(struct img_par *img);
633 void writeMotionInfo2NAL_Bframe(struct img_par *img, struct inp_par *
    inp);
634 int BlkSize2CodeNumber(int blc_size_h, int blc_size_v);
635
636
637 #endif
638
```

```

1  //
   *****
   *****
2  //
   *****
   *****
3  // Image.c   Code one image/slice
4  //
5  // Main contributors (see contributors.h for copyright, address and
   affiliation details)
6  //
7  // Inge Lille-Langøy           <inge.lille-langoy@telenor.com>
8  // Rickard Sjöberg            <rickard.sjoberg@era.ericsson.se>
9  // Jani Lainema               <jani.lainema@nokia.com>
10 // Sebastian Purreiter        <sebastian.purreiter@mch.siemens.de>
11 // Byeong-Moon Jeon           <jeonbm@lge.com>
12 // Yoon-Seong Soh             <yunsung@lge.com>
13 // Thomas Stockhammer         <stockhammer@ei.tum.de>
14 // Detlev Marpe               <marpe@hhi.de>
15 // Guido Heising              <heising@hhi.de>
16 //
   *****
   *****
17 //
   *****
   *****
18 #include "contributors.h"
19
20 #include <stdlib.h>
21 #include <stdio.h>
22 #include <math.h>
23 #include <time.h>
24 #include <sys/timeb.h>
25 #include <string.h>
26 #include <memory.h>
27
28 #include "global.h"
29 #include "elements.h"
30 #include "image.h"
31
32
33 /*****
   **
34  *
35  *   Name :           encode_one_frame()
36  *
37  *   Description: Encodes one frame
38  *
39  *****/
40 int encode_one_frame(struct img_par *img, struct inp_par *inp, struct
   stat_par *stat, struct snr_par *snr)
41 {
42     Boolean end_of_frame = FALSE;
43     const int total_number_mb = (img->width * img->height)/(
   MB_BLOCK_SIZE*MB_BLOCK_SIZE);
44     SyntaxElement imgTypeSymbol;
45
46     time_t ltime1;    // for time measurement
47     time_t ltime2;
48
49 #ifdef WIN32
50     struct _timeb tstruct1;
51     struct _timeb tstruct2;
52 #else
53     struct tm *l_time;

```

```

54     char string[20];
55     struct timeb tstruct1;
56     struct timeb tstruct2;
57     time_t now;
58 #endif
59
60     int tmp_time;
61
62 #ifdef WIN32
63     _ftime( &tstruct1 );    // start time ms
64 #else
65     ftime(&tstruct1);
66 #endif
67     time( &time1 );        // start time s
68
69
70     /* Initialize frame with all stat and img variables */
71     init_frame(img, inp, stat);
72
73     /* Initialize loopfilter */
74     init_loop_filter(img);
75
76     /* Select picture type: write coding to symbol */
77     select_picture_type(img, inp, &imgTypeSymbol);
78
79     /* Read one new frame */
80     read_one_new_frame(img, inp);
81
82     if (img->type == B_IMG)
83         Bframe_ctr++;
84
85     while (end_of_frame == FALSE)    /* loop over slices */
86     {
87         /* Encode the current slice */
88         encode_one_slice(img,inp,stat,&imgTypeSymbol);
89
90         /* Proceed to next slice */
91         img->current_slice_nr++;
92         stat->bit_slice = 0;
93         if (img->current_mb_nr == total_number_mb) /* end of frame
94             reached? */
95             end_of_frame = TRUE;
96     }
97
98     loopfilter(img); // reconstructed imgY, imgUV -> loop-filtered
99                     imgY, imgUV
100
101     // in future only one call of oneforthpix() for all frame tyoes
102     // will be necessary, because
103     // mref buffer will be increased by one frame to store also the
104     // next P-frame. Then mref_P
105     // will not be used any more
106     if (img->type != B_IMG) /*all I- and P-frames*/
107     {
108         if (inp->successive_Bframe == 0 || img->number == 0)
109             oneforthpix(img,inp); // I- and P-frames:loop-filtered
110             imgY, imgUV -> mref[][][], mcef[][][]
111         else
112             oneforthpix_2(img); // I- and P-frames prior a
113             B-frame:loop-filtered imgY, imgUV -> mref_P[][][],
114             mcef_P[][][]
115
116             // I- and P-frames prior a
117             B-frame:loop-filtered imgY, imgUV
118             -> mref_P[][][], mcef_P[][][]
119
120     }
121     else

```

```

111         if (img->b_frame_to_code == inp->successive_Bframe)
112             copy2mref(img);          // last successive B-frame:
113             mref_P[][][], mcef_P[][][] (loop-filtered imgY,
114             imgUV)-> mref[][][], mcef[][][]
115
116         find_snr(snr,img);
117
118         time(&lttime2);          // end time sec
119         #ifdef WIN32
120             _ftime(&tstruct2);    // end time ms
121         #else
122             ftime(&tstruct2);     // end time ms
123         #endif
124         tmp_time=(lttime2*1000+tstruct2.millitm) - (lttime1*1000+tstruct1.
125         millitm);
126         tot_time=tot_time + tmp_time;
127
128         /* Write reconstructed images */
129         write_reconstructed_image(img, inp);
130
131         if(img->number == 0)
132         {
133             printf("%3d(I) %8d %4d %7.4f %7.4f %7.4f %5d \n",
134             frame_no, stat->bit_ctr-stat->bit_ctr_n,
135             img->qp, snr->snr_y, snr->snr_u, snr->snr_v, tmp_time
136             );
137             stat->bitr0=stat->bitr;
138             stat->bit_ctr_0=stat->bit_ctr;
139             stat->bit_ctr=0;
140         }
141         else
142         {
143             if (img->type != B_IMG)
144             {
145                 stat->bit_ctr_P += stat->bit_ctr-stat->bit_ctr_n;
146                 printf("%3d(P) %8d %4d %7.4f %7.4f %7.4f %5d \n",
147                 frame_no, stat->bit_ctr-stat->bit_ctr_n,
148                 img->qp, snr->snr_y, snr->snr_u, snr->snr_v, tmp_time
149                 );
150             }
151             else
152             {
153                 stat->bit_ctr_B += stat->bit_ctr-stat->bit_ctr_n;
154                 printf("%3d(B) %8d %4d %7.4f %7.4f %7.4f %5d \n",
155                 frame_no, stat->bit_ctr-stat->bit_ctr_n, img->qp,
156                 snr->snr_y, snr->snr_u, snr->snr_v, tmp_time);
157             }
158         }
159         stat->bit_ctr_n=stat->bit_ctr;
160
161         if(img->number == 0)
162             return 0;
163         else
164             return 1;
165     }
166
167     /*****
168     *
169     *   Name :          encode_one_slice()
170     *
171     *   Description: Encodes one slice
172     *
173     *****/

```

```

171  */
172  void encode_one_slice(struct img_par *img, struct inp_par *inp, struct
173  stat_par *stat, SyntaxElement *imgTypeSym)
174  {
175      Boolean end_of_slice = FALSE;
176      Boolean recode_macroblock;
177      int len;
178
179      /* Initializes the parameters of the current slice */
180      init_slice(img, inp);
181
182      /* Write slice or picture header */
183      len = start_slice(img, inp, imgTypeSym);
184
185      /* Update statistics */
186      if (img->current_mb_nr > 0)
187      {
188          img->mb_data[img->current_mb_nr].bitcounter[BITS_HEADER_MB] =
189          len;
190      }
191      else
192      /* first slice includes picture header */
193      {
194          if (inp->symbol_mode == UVLC)
195          {
196              stat->bit_ctr += len;
197              stat->bit_slice += len;
198          }
199          stat->bit_use_head_mode[img->type] += len;
200      }
201
202      while (end_of_slice == FALSE) /* loop over macroblocks */
203      {
204          /* recode_macroblock is used in slice mode two and three
205          where */
206          /* backing of one macroblock in the bitstream is
207          possible */
208          recode_macroblock = FALSE;
209
210          /* Initializes the current macroblock */
211          start_macroblock(img, inp, stat);
212
213          /* Encode one macroblock */
214          encode_one_macroblock(img, inp, stat);
215
216          /* Pass the generated syntax elements to the NAL */
217          write_one_macroblock(img, inp, stat);
218
219          /* Terminate processing of the current macroblock */
220          terminate_macroblock(img, inp, stat, &end_of_slice, &
221          recode_macroblock);
222
223          if (recode_macroblock == FALSE) /* The final
224          processing of the macroblock has been done */
225              proceed2nextMacroblock(img, inp, stat); /* Go to next
226          macroblock */
227      }
228
229      terminate_slice(img, inp, stat);
230  }
231
232  /**
233  ****

```

```

227  *
228  *   Name :           init_frame()
229  *
230  *   Description: Initializes the parameters for a new frame
231  *
232  *****/
233 void init_frame(struct img_par *img, struct inp_par *inp, struct
stat_par *stat)
234 {
235     int i,j,k;
236     int prevP_no, nextP_no;
237
238     img->current_mb_nr=0;
239     img->current_slice_nr=0;
240     stat->bit_slice = 0;
241
242     img->mb_y = img->mb_x = 0;
243     img->block_y = img->pix_y = img->pix_c_y = 0;    /* define
vertical positions */
244     img->block_x = img->pix_x = img->block_c_x = img->pix_c_x = 0; /*
define horizontal positions */
245
246     if (inp->intra_upd > 0 && img->mb_y <= img->mb_y_intra)
247         img->height_err=(img->mb_y_intra*16)+15;    /* for extra
intra MB */
248     else
249         img->height_err=img->height-1;
250
251     if(img->type != B_IMG)
252     {
253         if (img->type == INTRA_IMG)
254             img->qp = inp->qp0;                    /* set quant.
parameter for I-frame */
255         else
256             img->qp = inp->qpN;                    /* set quant.
parameter for P-frame */
257
258         img->tr=img->number*(inp->jumpd+1);
259
260         if(img->number!=0 && inp->successive_Bframe != 0)    /* B
pictures to encode */
261             nextP_tr=img->tr;
262
263         img->mb_y_intra=img->mb_y_upd;    /* img->mb_y_intra
indicates which GOB to intra code for this frame */
264
265         if (inp->intra_upd > 0)                /* if error robustness, find
next GOB to update */
266         {
267             img->mb_y_upd=(img->number/inp->intra_upd) % (img->width/
MB_BLOCK_SIZE);
268         }
269     }
270     else
271     {
272         img->p_interval = inp->jumpd+1;
273         prevP_no = (img->number-1)*img->p_interval;
274         nextP_no = img->number*img->p_interval;
275
276
277         img->b_interval = (int)((float)(inp->jumpd+1)/(inp->
successive_Bframe+1.0)+0.5);
278
279         img->qp = inp->qpB;

```

```

280
281     img->tr= prevP_no+img->b_interval*img->b_frame_to_code; //
    from prev_P
282     if(img->tr >= nextP_no)
283         img->tr=nextP_no-1;
284
285     // initialize arrays
286     for(k=0; k<2; k++)
287         for(i=0; i<img->height/BLOCK_SIZE; i++)
288             for(j=0; j<img->width/BLOCK_SIZE+4; j++)
289                 {
290                     tmp_fwMV[k][i][j]=0;
291                     tmp_bwMV[k][i][j]=0;
292                     dfMV[k][i][j]=0;
293                     dbMV[k][i][j]=0;
294                 }
295     for(i=0; i<img->height/BLOCK_SIZE; i++)
296         for(j=0; j<img->width/BLOCK_SIZE; j++)
297             {
298                 fw_refFrArr[i][j]=bw_refFrArr[i][j]=-1;
299             }
300
301     }
302 }
303
304 /*****
305  *
306  *   Name :           init_slice()
307  *
308  *   Description: Initializes the parameters for a new slice
309  *
310  *****/
311 void init_slice(struct img_par *img, struct inp_par *inp)
312 {
313
314     int i;
315     Slice *curr_slice = img->currentSlice;
316     DataPartition *dataPart;
317     Bitstream *currStream;
318
319     curr_slice->picture_id = img->tr%256;
320     curr_slice->slice_nr = img->current_slice_nr;
321     curr_slice->qp = img->qp;
322     curr_slice->format = inp->img_format;
323     curr_slice->start_mb_nr = img->current_mb_nr;
324     curr_slice->dp_mode = inp->partition_mode;
325     curr_slice->slice_too_big = dummy_slice_too_big;
326
327     for (i=0; i<curr_slice->max_part_nr; i++)
328     {
329         dataPart = &(curr_slice->partArr[i]);
330
331         /* in principle it is possible to assign to each partition */
332         /* a different entropy coding method */
333         if (inp->symbol_mode == UVLC)
334             dataPart->writeSyntaxElement = writeSyntaxElement_UVLC;
335         else
336             dataPart->writeSyntaxElement = writeSyntaxElement_CABAC;
337         currStream = dataPart->bitstream;
338         currStream->bits_to_go = currStream->stored_bits_to_go;
339         currStream->byte_pos = currStream->stored_byte_pos;
340         currStream->byte_buf = currStream->stored_byte_buf;
341
342     }

```

```

343     }
344
345     /*****
346     *
347     *   Name :           select_picture_type()
348     *
349     *   Description: Selects picture type and codes it to symbol
350     *
351     *****/
352 void select_picture_type(struct img_par *img, struct inp_par *inp,
SyntaxElement *symbol)
353 {
354     int multpred;
355
356     if (inp->no_multpred <= 1)
357         multpred=FALSE;
358     else
359         multpred=TRUE;           /* multiple reference frames in
motion search */
360
361     if (img->type == INTRA_IMG)
362     {
363         symbol->len=3;
364         symbol->inf=1;
365         symbol->value1 = 2;
366     }
367     else if((img->type == INTER_IMG) && (multpred == FALSE)) /* inter
single reference frame */
368     {
369         symbol->len=1;
370         symbol->inf=0;
371         symbol->value1 = 0;
372     }
373     else if((img->type == INTER_IMG) && (multpred == TRUE)) /* inter
multiple reference frames */
374     {
375         symbol->len=3;
376         symbol->inf=0;
377         symbol->value1 = 1;
378     }
379     else if((img->type == B_IMG) && (multpred == FALSE))
380     {
381         symbol->len=5;
382         symbol->inf=0;
383         symbol->value1 = 3;
384     }
385     else if((img->type == B_IMG) && (multpred == TRUE))
386     {
387         symbol->len=5;
388         symbol->inf=1;
389         symbol->value1 = 4;
390     }
391     else
392     {
393         error("Picture Type not supported!");
394     }
395
396     #if TRACE
397         sprintf(symbol->tracestring, "Image type = %3d ", img->type);
398     #endif
399     symbol->type = SE_PTYPE;
400 }
401
402 /*****

```

```

403  **
404  *   Name :           init_loop_filter()
405  *
406  *   Description: initializes loop filter
407  *
408  ****
409  */
410  void init_loop_filter(struct img_par *img)
411  {
412      int i, j;
413
414      for (i=0;i<img->width/4+2;i++)
415          for (j=0;j<img->height/4+2;j++)
416              loopb[i+1][j+1]=0;
417
418      for (i=0;i<img->width_cr/4+2;i++)
419          for (j=0;j<img->height_cr/4+2;j++)
420              loopc[i+1][j+1]=0;
421  }
422
423  /*****
424  **
425  *   Name :           read_one_new frame()
426  *
427  *   Description: Reads new frame from file and sets frame_no
428  *
429  ****
430  */
431  void read_one_new_frame(struct img_par *img, struct inp_par *inp)
432  {
433      int i, j, uv;
434      int status; //frame_no;
435      int frame_size = img->height*img->width*3/2;
436
437      if(img->type == B_IMG)
438          frame_no = (img->number-1)*(inp->jumpd+1)+img->b_interval*img
439          ->b_frame_to_code;
440      else
441          frame_no = img->number*(inp->jumpd+1);
442
443      rewind (p_in);
444      status = fseek (p_in, frame_no * frame_size, 0);
445
446      if (status != 0)
447      {
448          sprintf(errortext, "Error in seeking frame no: %d\n", frame_no
449          );
450          error(errortext);
451      }
452
453      for (j=0; j < img->height; j++)
454          for (i=0; i < img->width; i++)
455              imgY_org[j][i]=fgetc(p_in);
456      for (uv=0; uv < 2; uv++)
457          for (j=0; j < img->height_cr ; j++)
458              for (i=0; i < img->width_cr; i++)
459                  imgUV_org[uv][j][i]=fgetc(p_in);
460  }
461
462  /*****
463  **
464  *
465  *   Name :           write_reconstructed_image()

```

```

462  *
463  *   Description: Writes reconstructed image(s) to file
464  *               This can be done more elegant!
465  *
466  *****/
467 void write_reconstructed_image(struct img_par *img, struct inp_par *
inp)
468 {
469     int i, j, k;
470
471     if(img->type != B_IMG)
472     {
473         // write reconstructed image (IPPP)
474         if(inp->write_dec && inp->successive_Bframe==0)
475         {
476             for (i=0; i < img->height; i++)
477                 for (j=0; j < img->width; j++)
478                     fputc(min(imgY[i][j],255),p_dec);
479
480             for (k=0; k < 2; ++k)
481                 for (i=0; i < img->height/2; i++)
482                     for (j=0; j < img->width/2; j++)
483                         fputc(min(imgUV[k][i][j],255),p_dec);
484         }
485
486         // write reconstructed image (IBPBP) : only intra written
487         else if (inp->write_dec && img->number==0 && inp->
successive_Bframe!=0)
488         {
489             for (i=0; i < img->height; i++)
490                 for (j=0; j < img->width; j++)
491                     fputc(min(imgY[i][j],255),p_dec);
492
493             for (k=0; k < 2; ++k)
494                 for (i=0; i < img->height/2; i++)
495                     for (j=0; j < img->width/2; j++)
496                         fputc(min(imgUV[k][i][j],255),p_dec);
497         }
498
499         // next P picture. This is saved with recon B picture after B
picture coding
500         if (inp->write_dec && img->number!=0 && inp->successive_Bframe
!=0)
501         {
502             for (i=0; i < img->height; i++)
503                 for (j=0; j < img->width; j++)
504                     nextP_imgY[i][j]=imgY[i][j];
505             for (k=0; k < 2; ++k)
506                 for (i=0; i < img->height/2; i++)
507                     for (j=0; j < img->width/2; j++)
508                         nextP_imgUV[k][i][j]=imgUV[k][i][j];
509         }
510     }
511     else
512     {
513         /* save B frame */
514         for (i=0; i < img->height; i++)
515             for (j=0; j < img->width; j++)
516                 fputc(min(imgY[i][j],255),p_dec);
517         for (k=0; k < 2; ++k)
518             for (i=0; i < img->height/2; i++)
519                 for (j=0; j < img->width/2; j++)
520                     fputc(min(imgUV[k][i][j],255),p_dec);
521
522         /* If this is last B frame also store P frame */

```

```

523         if(img->b_frame_to_code == inp->successive_Bframe)
524         {
525             /* save P picture */
526             for (i=0; i < img->height; i++)
527                 for (j=0; j < img->width; j++)
528                     fputc(min(nextP_imgY[i][j],255),p_dec);
529             for (k=0; k < 2; ++k)
530                 for (i=0; i < img->height/2; i++)
531                     for (j=0; j < img->width/2; j++)
532                         fputc(min(nextP_imgUV[k][i][j],255),p_dec);
533         }
534     }
535 }
536 }
537
538 /*****
539 *
540 *   Name :           oneforthpix()
541 *
542 *   Description: Upsample 4 times and store in buffer for multiple
543 *               reference frames
544 *
545 *   Changes: Thomas Wedi, 12.01.2001      <wedi@tnt.uni-hannover.de>
546 *
547 *               Slightly changed to avoid mismatches with the direct
548 *               interpolation filter in the decoder
549 *
550 *
551 *****/
552 void oneforthpix(struct img_par *img, struct inp_par *inp)
553 {
554     // int img4Y_tmp[576][704];
555     int is;
556     int i,j,i2,j2,j4;
557     int ie2,je2;
558     int uv;
559
560     img->frame_cycle=img->number % inp->no_multpred; /*GH
561     inp->no_multpred used insteadof MAX_MULT_PRED
562
563                                     frame buffer
564                                     size =
565                                     inp->no_multpred+
566                                     1*/
567
568     for (j=0; j < img->height; j++)
569     {
570         for (i=0; i < img->width; i++)
571         {
572             i2=i*2;
573             is=(ONE_FOURTH_TAP[0][0]*(imgY[j][i]          ]+imgY[j][min(
574                 img->width-1,i+1)])+
575                 ONE_FOURTH_TAP[1][0]*(imgY[j][max(0,i-1)]+imgY[j][min(
576                 img->width-1,i+2)])+
577                 ONE_FOURTH_TAP[2][0]*(imgY[j][max(0,i-2)]+imgY[j][min(
578                 img->width-1,i+3)]));
579             img4Y_tmp[2*j][i2  ]=imgY[j][i]*1024; /* 1/1 pix pos */
580             img4Y_tmp[2*j][i2+1]=is*32;          /* 1/2 pix pos */
581         }
582     }
583
584     for (i=0; i < img->width*2; i++)
585     {
586         for (j=0; j < img->height; j++)
587         {

```

```

580     j2=j*2;
581     j4=j*4;
582     /* change for TML4, use 6 TAP vertical filter */
583     is=(ONE_FOURTH_TAP[0][0]*(img4Y_tmp[j2      ][i]+
img4Y_tmp[min(2*img->height-2,j2+2)][i]))+
584         ONE_FOURTH_TAP[1][0]*(img4Y_tmp[max(0,j2-2)][i]+
img4Y_tmp[min(2*img->height-2,j2+4)][i]))+
585         ONE_FOURTH_TAP[2][0]*(img4Y_tmp[max(0,j2-4)][i]+
img4Y_tmp[min(2*img->height-2,j2+6)][i]))/32;
586
587     img4Y_tmp[j2+1][i]=is; /* 1/2 pix */
588     mref[img->frame_cycle][j4 ][i*2]=max(0,min(255,(int)((
img4Y_tmp[j2][i]+512)/1024))); /* 1/2 pix */
589     mref[img->frame_cycle][j4+2][i*2]=max(0,min(255,(int)((is+
512)/1024))); /* 1/2 pix */
590 }
591 }
592
593 /* 1/4 pix */
594 /* luma */
595 ie2=(img->width-1)*4;
596 je2=(img->height-1)*4;
597
598 for (j=0;j<je2+4;j+=2)
599     for (i=0;i<ie2+3;i+=2)
600         mref[img->frame_cycle][j][i+1]=max(0,min(255,(int)(img4Y_tmp[j/2
][i/2]+img4Y_tmp[j/2][min(ie2/2+1,i/2+1)]+1024)/2048));
601
602     for (i=0;i<ie2+4;i++)
603     {
604         for (j=0;j<je2+3;j+=2)
605         {
606             if( i%2 == 0 )
607                 mref[img->frame_cycle][j+1][i]=max(0,min(255,(int)
)(img4Y_tmp[j/2][i/2]+img4Y_tmp[min(je2/2+1,j/2+1
)][i/2+1024)/2048));
608             else
609                 mref[img->frame_cycle][j+1][i]=max(0,min(255,(int)(
img4Y_tmp[j/2      ][i/2      ] +
img4Y_tmp[min(je2/2+1,j/2+1)][i/2      ] +
img4Y_tmp[j/2      ][min(ie2/2+1,i/2+1)] +
img4Y_tmp[min(je2/2+1,j/2+1)][min(ie2/2+1,i/2+1)]
+ 2048)/4096));
614
615             /* "funny position" */
616             if( ((i&3) == 3)&&((j+1)&3) == 3))
617             {
618                 mref[img->frame_cycle][j+1][i] = (mref[img->
frame_cycle][j-2      ][i-3      ] +
619                     mref[img->frame_cycle][min(je2,j+2)][i-3
] +
620                     mref[img->frame_cycle][j-2      ][min(ie2,i
+1)] +
621                     mref[img->frame_cycle][min(je2,j+2)][min(ie2,i
+1)] + 2 )/4;
622             }
623         }
624     }
625
626     /* Chroma: */
627     for (uv=0; uv < 2; uv++)
628         for (j=0; j < img->height_cr; j++)
629             memcpy(mcef[img->frame_cycle][uv][j],imgUV[uv][j],img
->width_cr); /* just copy 1/1 pix, interpolate
"online" */
630

```

```

631     }
632
633     /*****
634     *   Name :   one_forthpix_2()
635     *
636     *   Description:
637     *       Upsample 4 times, store them in mref_P, mcef_P for next P
638     *       picture
639     *       and move them to mref, mcef
640     *
641     */
642
643     void oneforthpix_2(struct img_par *img)
644     {
645         // int img4Y_tmp[576][704];
646         int is;
647         int i,j,i2,j2,j4;
648         int ie2,je2;
649         int uv;
650
651         for (j=0; j < img->height; j++)
652         {
653             for (i=0; i < img->width; i++)
654             {
655                 i2=i*2;
656                 is=(ONE_FOURTH_TAP[0][0]*(imgY[j][i]          )+imgY[j][min(
657                     img->width-1,i+1)])+
658                     ONE_FOURTH_TAP[1][0]*(imgY[j][max(0,i-1)]+imgY[j][min(
659                     img->width-1,i+2)])+
660                     ONE_FOURTH_TAP[2][0]*(imgY[j][max(0,i-2)]+imgY[j][min(
661                     img->width-1,i+3)]));
662                 img4Y_tmp[2*j][i2]=imgY[j][i]*1024;    /* 1/1 pix pos */
663                 img4Y_tmp[2*j][i2+1]=is*32;            /* 1/2 pix pos */
664             }
665         }
666
667         for (i=0; i < img->width*2; i++)
668         {
669             for (j=0; j < img->height; j++)
670             {
671                 j2=j*2;
672                 j4=j*4;
673                 /* change for TML4, use 6 TAP vertical filter */
674                 is=(ONE_FOURTH_TAP[0][0]*(img4Y_tmp[j2]          ][i]+
675                     img4Y_tmp[min(2*img->height-2,j2+2)][i])+
676                     ONE_FOURTH_TAP[1][0]*(img4Y_tmp[max(0,j2-2)][i]+
677                     img4Y_tmp[min(2*img->height-2,j2+4)][i])+
678                     ONE_FOURTH_TAP[2][0]*(img4Y_tmp[max(0,j2-4)][i]+
679                     img4Y_tmp[min(2*img->height-2,j2+6)][i]))/32;
680
681                 img4Y_tmp[j2+1][i]=is;                /* 1/2 pix */
682                 mref_P[j4][i*2]=max(0,min(255,(int)((img4Y_tmp[j2][i]+
683                     512)/1024)));    /* 1/2 pix */
684                 mref_P[j4+2][i*2]=max(0,min(255,(int)((is+512)/1024)));
685                 /* 1/2 pix */
686             }
687         }
688
689         /* 1/4 pix */
690         /* luma */
691         ie2=(img->width-1)*4;
692         je2=(img->height-1)*4;
693
694         for (j=0;j<je2+4;j+=2)

```

```

686     for (i=0;i<ie2+3;i+=2)
687         mref_P[j][i+1]=max(0,min(255,(int)(img4Y_tmp[j/2][i/2]+img4Y_tmp
        [j/2][min(ie2/2+1,i/2+1)]+1024)/2048));
688
689     for (i=0;i<ie2+4;i++)
690     {
691         for (j=0;j<je2+3;j+=2)
692         {
693             if( i%2 == 0 )
694                 mref_P[j+1][i]=max(0,min(255,(int)(img4Y_tmp[j/2][
                i/2]+img4Y_tmp[min(je2/2+1,j/2+1)][i/2+1024)/2048
                ));
695             else
696                 mref_P[j+1][i]=max(0,min(255,(int)(
                img4Y_tmp[j/2][i/2] +
                img4Y_tmp[min(je2/2+1,j/2+1)][i/2] +
                img4Y_tmp[j/2][min(ie2/2+1,i/2+1)] +
                img4Y_tmp[min(je2/2+1,j/2+1)][min(ie2/2+1,i/2+1)]
                + 2048)/4096));
701
702             /* "funny position" */
703             if( ((i&3) == 3)&&((j+1)&3) == 3))
704             {
705                 mref_P[j+1][i] = (mref_P[j-2][i-3
                ] +
                mref_P[min(je2,j+2)][i-3] +
                mref_P[j-2][min(ie2,i+1)] +
                mref_P[min(je2,j+2)][min(ie2,i+1)] + 2 )/4;
706
707             }
708         }
709     }
710 }
711
712 /* Chroma: */
713 for (uv=0; uv < 2; uv++)
714     for (j=0; j < img->height_cr; j++)
715         memcpy(mcef_P[uv][j],imgUV[uv][j],img->width_cr); /*
716         just copy 1/1 pix, interpolate "online" */
717
718 }
719
720 /*****
721 *
722 * Name :      find_snr()
723 *
724 * Description: Find SNR for all three components
725 *
726 *****/
727 void find_snr(struct snr_par *snr,struct img_par *img)
728 {
729     int i,j;
730     int diff_y,diff_u,diff_v;
731     int impix;
732
733     /* Calculate PSNR for Y, U and V. */
734
735     /* Luma. */
736     impix = img->height*img->width;
737
738     diff_y=0;
739     for (i=0; i < img->width; ++i)
740     {
741         for (j=0; j < img->height; ++j)
742         {
743             diff_y += img->quad[abs(imgY_org[j][i]-imgY[j][i])];

```

```

744     }
745 }
746
747 /*      Chroma. */
748
749 diff_u=0;
750 diff_v=0;
751
752 for (i=0; i < img->width_cr; i++)
753 {
754     for (j=0; j < img->height_cr; j++)
755     {
756         diff_u += img->quad[abs(imgUV_org[0][j][i]-imgUV[0][j][i
757         diff_v += img->quad[abs(imgUV_org[1][j][i]-imgUV[1][j][i
758     }
759 }
760
761 /* Collecting SNR statistics */
762 if (diff_y != 0)
763 {
764     snr->snr_y=(float) (10*log10(65025*(float)impix/(float)diff_y
765     snr->snr_u=(float) (10*log10(65025*(float)impix/(float) (4*
766     snr->snr_v=(float) (10*log10(65025*(float)impix/(float) (4*
767 }
768
769 if (img->number == 0)
770 {
771     snr->snr_y1=(float) (10*log10(65025*(float)impix/(float)diff_y
772     snr->snr_u1=(float) (10*log10(65025*(float)impix/(float) (4*
773     snr->snr_v1=(float) (10*log10(65025*(float)impix/(float) (4*
774     snr->snr_ya=snr->snr_y1;
775     snr->snr_ua=snr->snr_u1;
776     snr->snr_va=snr->snr_v1;
777 }
778 /* B pictures */
779 else
780 {
781     snr->snr_ya=(float) (snr->snr_ya*(img->number+Bframe_ctr)+snr->
782     snr->snr_ua=(float) (snr->snr_ua*(img->number+Bframe_ctr)+snr->
783     snr->snr_va=(float) (snr->snr_va*(img->number+Bframe_ctr)+snr->
784 }
785 }
786
787
788 /*****
789 *
790 * Name :      loopfilter()
791 *
792 * Description: Filter to reduce blocking artifacts. The filter

```

```

strength
793 *          is QP dependent.
794 *
795 *****/
796
797 static unsigned char
798   overallActivity[256], qp_overallActivity = 255;
799 static int
800   beta;
801
802 extern const int QP2QUANT[32];
803
804 void loopfilter(struct img_par *img)
805 {
806   static int MAP[32];
807   /*GH: see dynamic mem allocation in
808    image.c */
808   /*byte imgY_tmp[288][352]; */ /* temp luma
809   image */
809   /*byte imgUV_tmp[2][144][176]; */ /* temp chroma
810   image */
810
811   int x,y,y4,x4,k,uv,str1,str2;
812
813   if (qp_overallActivity != img->qp)
814   {
815     int
816       i, alpha,
817       ALPHA_TABLE[32] = {
818         0, 0, 0, 0, 0, 0, 0, 0,
819         3, 3, 3, 7, 7, 7, 12, 17,
820         17, 22, 28, 34, 40, 47, 60, 67,
821         82, 89, 112, 137, 153, 187, 213, 249
822       },
823       BETA_TABLE[32] = {
824         0, 0, 0, 0, 0, 0, 0, 0,
825         3, 3, 3, 4, 4, 4, 6, 6,
826         6, 7, 8, 8, 9, 9, 10, 10,
827         11, 11, 12, 12, 13, 13, 14, 14
828       };
829
830     for (i=0; i<32; i++)
831       MAP[i] = QP2QUANT[i]-1;
832
833     /*
834     * ALPHA(img->qp) = ceil(2.0 * MAP(img->qp) * ln(MAP(img->qp)))
835     */
836     alpha = ALPHA_TABLE[img->qp];
837
838     /*
839     * BETA(img->qp) = floor(4.0 * log(MAP[img->qp]) + 0.5)
840     */
841     beta = BETA_TABLE[img->qp];
842
843     for (k=0; k<256; k++)
844       if (2*k >= alpha+(alpha/2))
845         overallActivity[k] = 1;
846       else if (2*k >= alpha)
847         overallActivity[k] = 2;
848       else
849         overallActivity[k] = 3;
850
851     qp_overallActivity = img->qp;
852
853   }

```

```

854
855     /* Luma hor */
856     for (y=0;y<img->height;y++)
857         for (x=0;x<img->width;x++)
858             imgY_tmp[y][x]=imgY[y][x];
859
860     for (y=0;y<img->height;y++)
861     {
862         y4=y/4;
863         for (x=4;x<img->width;x=x+4)
864         {
865             x4=x/4;
866
867             str1 = loopb[x4][y4+1];
868             str2 = loopb[x4+1][y4+1];
869
870             if((str1!=0)&&(str2!=0))
871             {
872                 /*
873                  * Copy one line of 4+4 reconstruction pels into the
874                  * filtering buffer.
875                  */
876                 for (k=0;k<8;k++)
877                     img->li[k]=imgY_tmp[y][x-4+k];
878
879                 /*
880                  * Check for longer filtering on macroblock border if either
881                  * block is intracoded.
882                  */
883                 if(!(x4%4) && (str1 == 3 || str2 == 3))
884                 {
885                     if(loop(img,str1,str2, 1, 0))
886                     {
887                         imgY_tmp[y][x-4+6] = img->lu[6];
888                         imgY[y][x-3] = img->lu[1];
889                         imgY[y][x+2] = img->lu[6];
890                     }
891                     else
892                         loop(img,str1,str2, 0, 0);
893
894                     /*
895                      * Place back the filtered values into the filtered
896                      * reconstruction buffer.
897                      */
898                     for (k=2;k<6;k++)
899                         imgY[y][x-4+k]=img->lu[k];
900                 }
901             }
902         }
903     }
904
905     /* Luma vert */
906     for (y=0;y<img->height;y++)
907         for (x=0;x<img->width;x++)
908             imgY_tmp[y][x]=imgY[y][x];
909
910     for (x=0;x<img->width;x++)
911     {
912         x4=x/4;
913         for (y=4;y<img->height;y=y+4)
914         {
915             y4=y/4;
916
917             str1 = loopb[x4+1][y4];
918             str2 = loopb[x4+1][y4+1];
919

```

```
917     if((str1 != 0) && (str2 != 0))
918     {
919         /*
920          * Copy one line of 4+4 reconstruction pels into the
921          * filtering buffer.
922          */
923         for(k=0;k<8;k++)
924             img->li[k]=imgY_tmp[y-4+k][x];
925
926         /*
927          * Check for longer filtering on macroblock border if either
928          * block is intracoded.
929          */
930         if(!(y4%4) && (str1 == 3 || str2 == 3))
931         {
932             if(loop(img,str1,str2, 1, 0)){
933                 imgY_tmp[y-4+6][x] = img->lu[6];
934                 imgY[y-3][x] = img->lu[1];
935                 imgY[y+2][x] = img->lu[6];
936             }
937         }
938         else
939             loop(img,str1,str2, 0, 0);
940
941         /*
942          * Place back the filtered values into the filtered
943          * reconstruction buffer.
944          */
945         for (k=2;k<6;k++)
946             imgY[y-4+k][x]=img->lu[k];
947     }
948 }
949
950 /* chroma */
951
952 for (uv=0;uv<2;uv++)
953 {
954     /* horizontal chroma */
955     for(y=0;y<img->height_cr;y++)
956         for(x=0;x<img->width_cr;x++)
957             imgUV_tmp[uv][y][x]=imgUV[uv][y][x];
958
959     for(y=0;y<img->height_cr;y++)
960     {
961         y4=y/4;
962         for(x=4;x<img->width_cr;x=x+4)
963         {
964             x4=x/4;
965
966             str1 = loopc[x4][y4+1];
967             str2 = loopc[x4+1][y4+1];
968
969             if((str1 != 0) && (str2 != 0))
970             {
971                 /*
972                  * Copy one line of 4+4 reconstruction pels into the
973                  * filtering buffer.
974                  */
975                 for(k=0;k<8;k++)
976                     img->li[k]=imgUV_tmp[uv][y][x-4+k];
977
978                 /*
979                  * Check for longer filtering on macroblock border if
980                  * either block is intracoded.
981                  */

```

```

978     if(!(x4%2) && (loopb[2*x4][2*y4+1] == 3 || loopb[2*x4+1][2*
979         y4+1] == 3))
980         loop(img, str1, str2, 1, 1);
981     else
982         loop(img, str1, str2, 0, 1);
983
984     /*
985     * Place back the filtered values into the filtered
986     reconstruction buffer.
987     */
988     for (k=2; k<6; k++)
989         imgUV[uv][y][x-4+k] = img->lu[k];
990 }
991 }
992
993 /* vertical chroma */
994 for (y=0; y<img->height_cr; y++)
995     for (x=0; x<img->width_cr; x++)
996         imgUV_tmp[uv][y][x] = imgUV[uv][y][x];
997
998 for (x=0; x<img->width_cr; x++)
999 {
1000     x4 = x/4;
1001     for (y=4; y<img->height_cr; y=y+4)
1002     {
1003         y4 = y/4;
1004
1005         str1 = loopc[x4+1][y4];
1006         str2 = loopc[x4+1][y4+1];
1007
1008         if((str1 != 0) && (str2 != 0))
1009         {
1010             /*
1011             * Copy one line of 4+4 reconstruction pels into the
1012             filtering buffer.
1013             */
1014             for (k=0; k<8; k++)
1015                 img->li[k] = imgUV_tmp[uv][y-4+k][x];
1016
1017             /*
1018             * Check for longer filtering on macroblock border if
1019             either block is intracoded.
1020             */
1021             if(!(y4%2) && (loopb[2*x4+1][2*y4] == 3 || loopb[2*x4+1][2*
1022                 y4+1] == 3))
1023                 loop(img, str1, str2, 1, 1);
1024             else
1025                 loop(img, str1, str2, 0, 1);
1026
1027             /*
1028             * Place back the filtered values into the filtered
1029             reconstruction buffer.
1030             */
1031             for (k=2; k<6; k++)
1032                 imgUV[uv][y-4+k][x] = img->lu[k];
1033         }
1034     }
1035 }
1036
1037 /*****
1038 **
1039 *
1040 * Name :      loop()

```

```
1037  *
1038  *   Description: Filters up to 6 pixels from 8 pixel input
1039  *
1040  *****/
1041  int loop(struct img_par *img, int ibl, int ibr, int longFilt, int
chroma)
1042  {
1043      int
1044          delta, halfLim, dr, dl, i, truncLimLeft, truncLimRight,
1045          diff, clip, clip_left, clip_right;
1046
1047      /* Limit the difference between filtered and nonfiltered based
1048       * on QP and strength
1049       */
1050      clip_left  = FILTER_STR[img->qp][ibl];
1051      clip_right = FILTER_STR[img->qp][ibr];
1052
1053      /* The step across the block boundaries */
1054      delta  = abs(img->li[3]-img->li[4]);
1055
1056      /* Find overall activity parameter (n/2) */
1057      halfLim = overallActivity[delta];
1058
1059      /* Truncate left limit to 2 for small stengths */
1060      if (ibl <= 1)
1061          truncLimLeft = (halfLim > 2) ? 2 : halfLim;
1062      else
1063          truncLimLeft = halfLim;
1064
1065      /* Truncate right limit to 2 for small stengths */
1066      if (ibr <= 1)
1067          truncLimRight = (halfLim > 2) ? 2 : halfLim;
1068      else
1069          truncLimRight = halfLim;
1070
1071      /* Find right activity parameter dr */
1072      for (dr=1; dr<truncLimRight; dr++)
1073      {
1074          if (dr*abs(img->li[4]-img->li[4+dr]) > beta)
1075              break;
1076      }
1077
1078      /* Find left activity parameter dl */
1079      for (dl=1; dl<truncLimLeft; dl++)
1080      {
1081          if (dl*abs(img->li[3]-img->li[3-dl]) > beta)
1082              break;
1083      }
1084
1085      if(dr < 2 || dl < 2)
1086      {
1087          /* no filtering when either one of activity params is below two */
1088          img->lu[2] = img->li[2];
1089          img->lu[3] = img->li[3];
1090          img->lu[4] = img->li[4];
1091          img->lu[5] = img->li[5];
1092
1093          return 0;
1094      }
1095
1096      if(longFilt)
1097      {
1098          if(dr == 3 && dl == 3 &&
1099             delta >= 2 && delta <  img->qp/4)
1100          {
```

```

1101         if(chroma)
1102         {
1103             img->lu[3] = (25*(img->li[1] + img->li[5]) + 26*(img->li[2] +
1104             img->li[3] + img->li[4]) + 64) >> 7;
1105             img->lu[2] = (25*(img->li[0] + img->li[4]) + 26*(img->li[1] +
1106             img->li[2] + img->lu[3]) + 64) >> 7;
1107             img->lu[4] = (25*(img->li[2] + img->li[6]) + 26*(img->li[3] +
1108             img->li[4] + img->li[5]) + 64) >> 7;
1109             img->lu[5] = (25*(img->li[3] + img->li[7]) + 26*(img->lu[4] +
1110             img->li[5] + img->li[6]) + 64) >> 7;
1111
1112             return 0;
1113         }
1114     else
1115     {
1116         img->lu[3] = (25*(img->li[1] + img->li[5]) + 26*(img->li[2] +
1117         img->li[3] + img->li[4]) + 64) >> 7;
1118         img->lu[2] = (25*(img->li[0] + img->li[4]) + 26*(img->li[1] +
1119         img->li[2] + img->lu[3]) + 64) >> 7;
1120         img->lu[1] = (25*(img->li[1] + img->lu[3]) + 26*(img->li[0] +
1121         img->li[1] + img->lu[2]) + 64) >> 7;
1122         img->lu[4] = (25*(img->li[2] + img->li[6]) + 26*(img->li[3] +
1123         img->li[4] + img->li[5]) + 64) >> 7;
1124         img->lu[5] = (25*(img->li[3] + img->li[7]) + 26*(img->lu[4] +
1125         img->li[5] + img->li[6]) + 64) >> 7;
1126         img->lu[6] = (25*(img->lu[4] + img->li[6]) + 26*(img->lu[5] +
1127         img->li[6] + img->li[7]) + 64) >> 7;
1128
1129         return 1;
1130     }
1131 }
1132
1133 /* Filter pixels at the edge */
1134 img->lu[4] = (21*(img->li[3] + img->li[5]) + 22*img->li[4] + 32) >>
1135 6;
1136 img->lu[3] = (21*(img->li[2] + img->li[4]) + 22*img->li[3] + 32) >>
1137 6;
1138
1139 if(dr == 3)
1140     img->lu[5] = (21*(img->lu[4] + img->li[6]) + 22*img->li[5] + 32)
1141     >> 6;
1142 else
1143     img->lu[5] = img->li[5];
1144
1145 if(dl == 3)
1146     img->lu[2] = (21*(img->li[1] + img->lu[3]) + 22*img->li[2] + 32)
1147     >> 6;
1148 else
1149     img->lu[2] = img->li[2];
1150
1151 /* Clipping parameter depends on one table and left and right act
1152 params */
1153 clip = (clip_left + clip_right + dl + dr) / 2;
1154
1155 /* Pixels at the edge are clipped differently */
1156 for (i=3; i<=4; i++)
1157 {
1158     diff = (int)img->lu[i] - (int)img->li[i];
1159
1160     if (diff)
1161     {
1162         if (diff > clip)
1163             diff = clip;
1164         else if (diff < -clip)
1165             diff = -clip;
1166     }
1167 }

```

```
1152         img->lu[i] = img->li[i] + diff;
1153     }
1154 }
1155
1156 /* pixel from left is clipped */
1157 diff = (int)img->lu[2] - (int)img->li[2];
1158
1159 if (diff)
1160 {
1161     if (diff > clip_left)
1162         diff = clip_left;
1163     else if (diff < -clip_left)
1164         diff = -clip_left;
1165
1166     img->lu[2] = img->li[2] + diff;
1167 }
1168
1169 /* pixel from right is clipped */
1170 diff = (int)img->lu[5] - (int)img->li[5];
1171
1172 if (diff)
1173 {
1174     if (diff > clip_right)
1175         diff = clip_right;
1176     else if (diff < -clip_right)
1177         diff = -clip_right;
1178
1179     img->lu[5] = img->li[5] + diff;
1180 }
1181
1182 return 0;
1183 }
1184
1185 /*****
1186  *
1187  *   Name :          dummy_slice_too_big()
1188  *
1189  *   Description: Just a placebo
1190  *
1191  * *****/
1192
1193 Boolean dummy_slice_too_big(int bits_slice)
1194 {
1195     return FALSE;
1196 }
```

```

1  //
   *****
   *****
2  //
   *****
   *****
3  // Lencod.c  H.26L encoder project main
4  //
5  // Main contributors (see contributors.h for copyright, address and
   affiliation details)
6  //
7  // Inge Lille-Langøy                <inge.lille-langoy@telenor.com>
8  // Rickard Sjöberg                 <rickard.sjoberg@era.ericsson.se>
9  // Stephan Wenger                  <stewe@cs.tu-berlin.de>
10 // Jani Lainema                    <jani.lainema@nokia.com>
11 // Byeong-Moon Jeon                <jeonbm@lge.com>
12 // Yoon-Seong Soh                  <yunsung@lge.com>
13 // Thomas Stockhammer              <stockhammer@ei.tum.de>
14 // Detlev Marpe                    <marpe@hhi.de>
15 // Guido Heising                   <heising@hhi.de>
16 //
   *****
   *****
17 //
   *****
   *****
18 #include "contributors.h"
19
20 #include <string.h>
21 #include <stdio.h>
22 #include <math.h>
23 #include <time.h>
24 #include <sys/timeb.h>
25 #include <stdlib.h>
26
27 #include "global.h"
28 #include "elements.h"
29
30 #define TML      "5"
31 #define VERSION  "5.996"
32
33 void main(int argc, char **argv)
34 {
35     struct inp_par    *inp;          /* input parameters from input
   configuration file */
36     struct stat_par   *stat;         /*
   statistics */
37     struct snr_par     *snr;         /* SNR
   values */
38     struct img_par     *img;         /* image
   parameters */
39
40     /* allocate memory */
41     inp = (struct inp_par *) calloc(1, sizeof(struct inp_par));
42     stat = (struct stat_par *) calloc(1, sizeof(struct stat_par));
43     snr = (struct snr_par *) calloc(1, sizeof(struct snr_par));
44     img = (struct img_par *) calloc(1, sizeof(struct img_par));
45
46     /* Read Configuration File */
47     if (argc != 2)
48     {
49         printf("Usage: %s <config.dat> \n", argv[0]);
50         printf("\t<config.dat> defines encoder parameters\n");
51         exit(-1);
52     }
53

```

```

54     /* Initializes Configuration Parameters with configuration file */
55     init_conf(inp, argv[1]);
56
57     /* Initialize Image Parameters */
58     init_img(inp, img);
59
60     /* Allocate Slice data struct */
61     malloc_slice(inp, img);
62
63     /* Initialize Statistic Parameters */
64     init_stat(inp, stat);
65
66     /* allocate memory for frame buffers*/
67     get_mem4global_buffers(inp, img);
68
69     /* Just some information which goes to the standard output */
70     information_init(inp, argv[1]);
71
72     /* Write sequence header; open bitstream files */
73     start_sequence(inp);
74
75     /* B pictures */
76     Bframe_ctr=0;
77     tot_time=0;                // time for total encoding session
78
79     for (img->number=0; img->number < inp->no_frames; img->number++)
80     {
81         if (img->number == 0)
82             img->type = INTRA_IMG;                /* set image type for
83             first image to I-frame */
84         else
85             img->type = INTER_IMG;                /* P-frame */
86
87         encode_one_frame(img, inp, stat, snr); /* encode one I- or
88         P-frame */
89
90         if ((inp->successive_Bframe != 0) && (img->number > 0)) /*
91         B-frame(s) to encode */
92         {
93             img->type = B_IMG;                /* set image type
94             to B-frame */
95             for (img->b_frame_to_code=1; img->b_frame_to_code<=inp->
96             successive_Bframe; img->b_frame_to_code++)
97                 encode_one_frame(img, inp, stat, snr); /* encode
98                 one B-frame */
99         }
100     }
101
102     /* terminate sequence */
103     terminate_sequence(img, inp);
104
105     /* report everything */
106     report(inp, img, stat, snr);
107
108     free_slice(inp, img);
109
110     /* free allocated memory for frame buffers*/
111     free_mem4global_buffers(inp, img);
112
113     free(snr);
114     free(stat);
115     free(img);
116     free(inp);
117 }
118
119 /*****

```

```

114  *
115  *   Name :          init_conf(struct inp_par *inp, char *config_filename)
116  *
117  *   Description: Read input from configuration file
118  *
119  *   Input      : Name of configuration filename
120  *
121  *   Output     : none
122  *
123  *****/
124
125 void init_conf(struct inp_par *inp, char *config_filename)
126 {
127     FILE *fd;
128     int bck_tmp, NAL_mode;
129     int i,j;
130
131     for (i=0;i<8;i++)
132         for (j=0;j<2;j++)
133             inp->blc_size[i][j]=0;          /* init block size array */
134
135     if((fd=fopen(config_filename,"r")) == NULL)          /* read the
decoder configuration file */
136     {
137         printf("Error: Control file %s not found\n",config_filename);
138         exit(0);
139     }
140
141
142     /* read input parameters */
143     fscanf(fd,"%ld",&inp->no_frames);          /* number of
frames to be encoded */
144     fscanf(fd,"%*[^\\n]");          /* discard
everything to the beginning of ewnt line */
145
146     fscanf(fd,"%ld",&inp->qp0);          /* QP of first
frame (max 31) */
147     fscanf(fd,"%*[^\\n]");
148
149     if (inp->qp0 > 31 && inp->qp0 <= 0)
150     {
151         printf("Error input parameter quant_0,check configuration
file\\n");
152         exit (0);
153     }
154
155     fscanf(fd,"%ld",&inp->qpN);          /* QP of
remaining frames (max 31) */
156     fscanf(fd,"%*[^\\n]");
157     if (inp->qpN > 31 && inp->qpN <= 0)
158     {
159         printf("Error input parameter quant_n,check configuration
file\\n");
160         exit (0);
161     }
162
163     fscanf(fd,"%ld",&inp->jumpd);          /* number of
frames to skip in input sequence (e.g 2 takes frame 0,3,6,9...) */
164     fscanf(fd,"%*[^\\n]");
165
166     fscanf(fd,"%ld",&inp->hadamard);          /* 0:
'normal' SAD in 1/3 pixel search. 1: use 4x4 Hadamard transform
and 'Sum of absolute transform differens' in 1/3 pixel search */
167     fscanf(fd,"%*[^\\n]");

```

```

168
169     fscanf(fd,"%ld",&inp->search_range);
170
171     /* search range-integer pel search and 16x16 blocks. The search
172     window is generally around the predicted vector.
173     Max vector is 2xsearch_range. For 8x8 and 4x4 block sizes the
174     search range is 1/2 of that for 16x16 blocks. */
175
176     fscanf(fd,"%*[\n]");
177     if (inp->search_range > 16)
178     {
179         printf("Error in input parameter search_range, check
180         configuration file\n");
181         exit (0);
182     }
183
184     fscanf(fd,"%ld",&inp->no_multpred); /* 1:
185     prediction from the last frame only. 2: prediction from the last
186     or second last frame etc. Maximum 5 frames */
187     fscanf(fd,"%*[\n]");
188     if(inp->no_multpred > MAX_MULT_PRED)
189     {
190         printf("Error : input parameter 'no_multpred' exceeds limit
191         (1...5), check configuration file \n");
192         exit(0);
193     }
194
195     fscanf(fd,"%ld",&inp->img_format); /* GH:
196     0=SUB_QCIF, 1=QCIF; 2=CIF, 3=CIF_4. 4=CIF_16, 5=CUSTOM */
197     fscanf(fd,"%*[\n]");
198     if (inp->img_format != SUB_QCIF && inp->img_format != QCIF && inp->
199     img_format != CIF && inp->img_format != CIF_4 && inp->img_format !=
200     CIF_16 && inp->img_format != CUSTOM)
201     {
202         printf("Unsupported image format=%d,use:\nSub-QCIF=0, QCIF=1,
203         CIF=2, 4CIF=3, 16CIF=4 or CUSTOM=5\n",inp->img_format);
204         exit(0);
205     }
206
207     fscanf(fd,"%ld",&inp->img_width); /* GH: if
208     CUSTOM picture format, width and height will be used, else
209     ignored*/
210     fscanf(fd,"%*[\n]");
211     fscanf(fd,"%ld",&inp->img_height); /* GH: size
212     must be a multiple of 16 pels */
213     fscanf(fd,"%*[\n]");
214     if (inp->img_format == CUSTOM && (((inp->img_height % 16) != 0) ||
215     ((inp->img_width % 16) != 0)))
216     {
217         printf("Unsupported CUSTOM image format x=%d,y=%d, must be a
218         multiple of 16\n",inp->img_width,inp->img_height);
219         exit(0);
220     }
221
222     fscanf(fd,"%ld",&inp->yuv_format); /* GH: YUV
223     format (0=4:0:0, 1=4:2:0, 2=4:2:2, 3=4:4:4,currently only 4:2:0
224     is supported)*/
225     fscanf(fd,"%*[\n]");
226     if (inp->yuv_format != 1)
227     {
228         printf("Unsupported YUV format=%d, must be 1 (YUV=4:2:0)\n",
229         inp->yuv_format);
230         exit(0);
231     }
232
233     fscanf(fd,"%ld",&inp->color_depth); /* GH: YUV

```

```

        color depth per component in bit/pel (currently only 8 bit/pel is
        supported)*/
216     fscanf(fd,"%*[^\\n]");
217     if (inp->color_depth != 8)
218     {
219         printf("Unsupported color depth %d bit/pel, currently only 8
        bit/pel is available\\n",inp->color_depth);
220         exit(0);
221     }
222
223     fscanf(fd,"%ld",&inp->intra_upd);          /* For error
        robustness.
224
        0: no
        special action.
225
        1: One
        GOB/frame is intra coded as regular 'update'.
226
        2: One
        GOB every 2 frames is intra coded etc.
227
        In
        connection with this intra update,restrictions is put on
228
        motion
        vectors to prevent errors to propagate from the past */
229     fscanf(fd,"%*[^\\n]");
230     fscanf(fd,"%ld",&inp->write_dec);          /* 0: Do not write
        decoded image to file, 1: Write decoded image to file */
231
232     /* read block sizes for motion search */
233     fscanf(fd,"%*[^\\n]");
234     fscanf(fd,"%ld",&bck_tmp);
235     if (bck_tmp)
236     {
237         inp->bkc_size[1][0]=16;
238         inp->bkc_size[1][1]=16;
239     }
240     fscanf(fd,"%ld",&bck_tmp);
241     fscanf(fd,"%*[^\\n]");
242     fscanf(fd,"%ld",&bck_tmp);
243     if (bck_tmp)
244     {
245         inp->bkc_size[2][0]=16;
246         inp->bkc_size[2][1]= 8;
247     }
248     fscanf(fd,"%*[^\\n]");
249     fscanf(fd,"%ld",&bck_tmp);
250     if (bck_tmp)
251     {
252         inp->bkc_size[3][0]= 8;
253         inp->bkc_size[3][1]=16;
254     }
255     fscanf(fd,"%*[^\\n]");
256     fscanf(fd,"%ld",&bck_tmp);
257     if (bck_tmp)
258     {
259         inp->bkc_size[4][0]= 8;
260         inp->bkc_size[4][1]= 8;
261     }
262     fscanf(fd,"%*[^\\n]");
263     fscanf(fd,"%ld",&bck_tmp);
264     if (bck_tmp)
265     {
266         inp->bkc_size[5][0]= 8;
267         inp->bkc_size[5][1]= 4;
268     }
269     fscanf(fd,"%*[^\\n]");
270     fscanf(fd,"%ld",&bck_tmp);
271     if (bck_tmp)

```

```
272     {
273         inp->blc_size[6][0]= 4;
274         inp->blc_size[6][1]= 8;
275     }
276     fscanf(fd,"%*[^\\n]");
277     fscanf(fd,"%ld",&bck_tmp);
278     if (bck_tmp)
279     {
280         inp->blc_size[7][0]= 4;
281         inp->blc_size[7][1]= 4;
282     }
283     fscanf(fd,"%*[^\\n]");
284
285     fscanf(fd,"%s",inp->infile);    /* YUV 4:2:0 input format */
286     fscanf(fd,"%*[^\\n]");
287
288     fscanf(fd,"%s",inp->outfile);    /* H.26L compressed output
289     bitsream */
290     fscanf(fd,"%*[^\\n]");
291
292     if ((p_in=fopen(inp->infile,"rb"))==0)
293     {
294         printf("Input file %s does not exist \\n",inp->infile);
295         exit(0);
296     }
297     if ((p_dec=fopen("test.dec", "wb"))==0)    /* decodet picture,for
298     debugging */
299     {
300         printf("Error open file test.dec \\n");
301         exit(0);
302     }
303     #if TRACE
304     if ((p_trace=fopen("trace_enc.txt","w"))==0)
305     {
306         printf("Error open file trace_enc.txt\\n");
307         exit(0);
308     }
309     #endif
310
311     fscanf(fd,"%d",&(NAL_mode));    /* NAL mode */
312     fscanf(fd,"%*[^\\n]");
313
314     switch(NAL_mode)
315     {
316     case 0:
317         inp->of_mode = PAR_OF_26L;
318         /* Note: Data Partitioning in 26L File Format not yet
319         supported */
320         inp->partition_mode = PAR_DP_1;
321         break;
322     case 1:
323         inp->of_mode = PAR_OF_SLICE;
324         inp->partition_mode = PAR_DP_1;
325         break;
326     case 2:
327         inp->of_mode = PAR_OF_SLICE;
328         inp->partition_mode = PAR_DP_2;
329         break;
330     case 3:
331         inp->of_mode = PAR_OF_SLICE;
332         inp->partition_mode = PAR_DP_4;
333         break;
334     case 4:
335         inp->of_mode = PAR_OF_NAL;
336         inp->partition_mode = PAR_DP_1;
```

```

335         break;
336     case 5:
337         inp->of_mode = PAR_OF_NAL;
338         inp->partition_mode = PAR_DP_2;
339         break;
340     case 6:
341         inp->of_mode = PAR_OF_NAL;
342         inp->partition_mode = PAR_DP_4;
343         break;
344     default:
345         printf("NAL mode %i is not supported\n", NAL_mode);
346         exit(1);
347     }
348
349
350     fscanf(fd, "%d", &(inp->slice_mode));
351     fscanf(fd, "%*[^\\n]");
352
353     fscanf(fd, "%d", &(inp->slice_argument));
354     fscanf(fd, "%*[^\\n]");
355
356
357     /* B pictures */
358     fscanf(fd, "%ld", &inp->successive_Bframe); /* 0: B frame is
not added */
359     fscanf(fd, "%*[^\\n]");
360     fscanf(fd, "%ld", &inp->qpB); /* QP of B frames
(max 31) */
361     fscanf(fd, "%*[^\\n]");
362     if (inp->qpB > 31 && inp->qpB <= 0)
363     {
364         printf("Error input parameter quant_B, check configuration
file\\n");
365         exit (0);
366     }
367     if(inp->successive_Bframe > inp->jumpd)
368     {
369         sprintf(errortext, "Warning !! Number of B-frames %d can not
exceed the number of frames skipped\\n",
inp->successive_Bframe);
370         error(errortext);
371     }
372 }
373
374     /* Symbol mode */
375     fscanf(fd, "%ld", &inp->symbol_mode); /* 0:
UVLC 1: CABAC */
376     fscanf(fd, "%*[^\\n]");
377     if (inp->symbol_mode != UVLC && inp->symbol_mode != CABAC)
378     {
379         sprintf(errortext, "Unsupported symbol mode=%d, use UVLC=0 or
CABAC=1\\n", inp->symbol_mode);
380         error(errortext);
381     }
382 }
383
384 /*****
**
385 *
386 * Name : void init_img()
387 *
388 * Description: Initializes the Image structure with appropriate
parameters
389 *
390 * Input : Input Parameters struct inp_par *inp
391 *
392 * Output : Image Parameters struct img_par *img

```

```
393  *
394  *****/
395  void init_img(struct inp_par *inp, struct img_par *img)
396  {
397      int i,j;
398
399      img->no_multpred=inp->no_multpred;
400      img->framerate=INIT_FRAME_RATE; /* The basic frame rate (of the
401                                       original sequence) */
402
403      if (inp->img_format == SUB_QCIF)
404      {
405          img->width      = 128;
406          img->height     = 96;
407          img->width_cr   = 64;
408          img->height_cr  = 48;
409      }
410      else if (inp->img_format == QCIF)
411      {
412          img->width      = 176;
413          img->height     = 144;
414          img->width_cr   = 88;
415          img->height_cr  = 72;
416      }
417      else if (inp->img_format == CIF)
418      {
419          img->width      = 352;
420          img->height     = 288;
421          img->width_cr   = 176;
422          img->height_cr  = 144;
423      }
424      else if (inp->img_format == CIF_4)
425      {
426          img->width      = 704;
427          img->height     = 576;
428          img->width_cr   = 352;
429          img->height_cr  = 288;
430      }
431      else if (inp->img_format == CIF_16)
432      {
433          img->width      = 1408;
434          img->height     = 1152;
435          img->width_cr   = 704;
436          img->height_cr  = 576;
437      }
438      else /*CUSTOM*/
439      {
440          img->width      = inp->img_width;
441          img->height     = inp->img_height;
442          img->width_cr   = inp->img_width/2;
443          img->height_cr  = inp->img_height/2;
444      }
445
446      if(((img->mb_data) = (Macroblock *) calloc((img->width/
447          MB_BLOCK_SIZE) * (img->height/MB_BLOCK_SIZE),sizeof(Macroblock
448          ))) == NULL)
449          no_mem_exit(1);
450
451      if(((img->slice_numbers) = (int *) calloc((img->width/
452          MB_BLOCK_SIZE) * (img->height/MB_BLOCK_SIZE),sizeof(int))) ==
453          NULL)
454          no_mem_exit(1);
455
456      init(img);
```

```

453
454     /* allocate memory for intra pred mode buffer for each block:
img->ipredmode*/
455     /* int  img->ipredmode[90][74]; */
456     get_mem2Dint(&(img->ipredmode), img->width/BLOCK_SIZE+2, img->
height/BLOCK_SIZE+2);
457
458     /* Prediction mode is set to -1 outside the frame, indicating
that no prediction can be made from this part*/
459     for (i=0; i < img->width/BLOCK_SIZE+1; i++)
460     {
461         img->ipredmode[i+1][0]=-1;
462     }
463     for (j=0; j < img->height/BLOCK_SIZE+1; j++)
464     {
465         img->ipredmode[0][j+1]=-1;
466     }
467
468     img->mb_y_upd=0;
469
470 }
471
472 /*****
**
473 *
474 *   Name :          void malloc_slice()
475 *
476 *   Description: Allocates the slice structure along with its dependent
477 *                 data structures
478 *
479 *   Input      : Input Parameters struct inp_par *inp,  struct img_par
*img
480 *
481 *****/
482 void malloc_slice(struct inp_par *inp, struct img_par *img)
483 {
484     int i;
485     DataPartition *dataPart;
486     Slice *currSlice;
487     const int buffer_size = (img->width * img->height * 3)/2; /* DM
070301: The only assumption here is that we */
488
489                                     /* do not
consider the case of data expansion. */
490
491                                     /* So this is a
strictly conservative estimation */
492
493                                     /* of the size of
the bitstream buffer for one frame */
494
495                                     /* ~DM */
496
497     switch(inp->of_mode) /* init depending on NAL mode */
498     {
499     case PAR_OF_26L:
500         /* Current File Format */
501         img->currentSlice = (Slice *) calloc(1, sizeof(Slice));
502         if ( (currSlice = img->currentSlice) == NULL)
503         {
504             fprintf(stderr, "Memory allocation for Slice
datastruct in NAL-mode %d failed", inp->of_mode);
505             exit(1);
506         }
507         if (inp->symbol_mode == CABAC)

```

```

503     {
504         /* create all context models */
505         currSlice->mot_ctx = create_contexts_MotionInfo();
506         currSlice->tex_ctx = create_contexts_TextureInfo();
507     }
508
509     switch(inp->partition_mode)
510     {
511     case PAR_DP_1:
512         currSlice->max_part_nr = 1;
513         break;
514     case PAR_DP_2:
515         error("Data Partitioning Mode 2 in 26L-Format not yet
516             supported");
517         break;
518     case PAR_DP_4:
519         error("Data Partitioning Mode 4 in 26L-Format not yet
520             supported");
521         break;
522     default:
523         error("Data Partitioning Mode not supported!");
524         break;
525     }
526
527     currSlice->partArr = (DataPartition *) calloc(1, sizeof(
528         DataPartition));
529     if (currSlice->partArr == NULL)
530     {
531         fprintf(stderr, "Memory allocation for Data Partition
532             datastruct in NAL-mode %d failed", inp->of_mode);
533         exit(1);
534     }
535     dataPart = currSlice->partArr;
536     dataPart->bitstream = (Bitstream *) calloc(1, sizeof(
537         Bitstream));
538     if (dataPart->bitstream == NULL)
539     {
540         fprintf(stderr, "Memory allocation for Bitstream
541             datastruct in NAL-mode %d failed", inp->of_mode);
542         exit(1);
543     }
544     dataPart->bitstream->streamBuffer = (byte *) calloc(
545         buffer_size, sizeof(byte));
546     if (dataPart->bitstream->streamBuffer == NULL)
547     {
548         fprintf(stderr, "Memory allocation for bitstream
549             buffer in NAL-mode %d failed", inp->of_mode);
550         exit(1);
551     }
552     /* Initialize storage of bitstream parameters */
553     dataPart->bitstream->stored_bits_to_go = 8;
554     dataPart->bitstream->stored_byte_pos = 0;
555     dataPart->bitstream->stored_byte_buf = 0;
556     return;
557 case PAR_OF_SLICE:
558     /* NAL File Format */
559     img->currentSlice = (Slice *) calloc(1, sizeof(Slice));
560     if ( (currSlice = img->currentSlice) == NULL)
561     {
562         fprintf(stderr, "Memory allocation for Slice
563             datastruct in NAL-mode %d failed", inp->of_mode);
564         exit(1);
565     }

```

```

560     switch(inp->partition_mode)
561     {
562     case PAR_DP_1:
563         currSlice->max_part_nr = 1;
564         break;
565     case PAR_DP_2:
566         currSlice->max_part_nr = 8;
567         break;
568     case PAR_DP_4:
569         currSlice->max_part_nr = 4;
570         break;
571     default:
572         error("Data Partitioning Mode not supported!");
573         break;
574     }
575
576     currSlice->partArr = (DataPartition *) calloc(currSlice->
max_part_nr, sizeof(DataPartition));
577     if (currSlice->partArr == NULL)
578     {
579         fprintf(stderr, "Memory allocation for Data Partition
datastruct in NAL-mode %d failed", inp->of_mode);
580         exit(1);
581     }
582
583     for (i=0; i<currSlice->max_part_nr; i++) /* loop over all
data partitions */
584     {
585         dataPart = &(currSlice->partArr[i]);
586         dataPart->bitstream = (Bitstream *) calloc(1, sizeof(
Bitstream));
587         if (dataPart->bitstream == NULL)
588         {
589             fprintf(stderr, "Memory allocation for Bitstream
datastruct in NAL-mode %d failed", inp->of_mode);
590             exit(1);
591         }
592         dataPart->bitstream->streamBuffer = (byte *) calloc(
buffer_size, sizeof(byte));
593         if (dataPart->bitstream->streamBuffer == NULL)
594         {
595             fprintf(stderr, "Memory allocation for bitstream
buffer in NAL-mode %d failed", inp->of_mode);
596             exit(1);
597         }
598         /* Initialize storage of bitstream parameters */
599         dataPart->bitstream->stored_bits_to_go = 8;
600         dataPart->bitstream->stored_byte_pos = 0;
601         dataPart->bitstream->stored_byte_buf = 0;
602     }
603     return;
604 case PAR_OF_NAL:
605     /* NAL File Format */
606     img->currentSlice = (Slice *) calloc(1, sizeof(Slice));
607     if ( (currSlice = img->currentSlice) == NULL)
608     {
609         fprintf(stderr, "Memory allocation for Slice
datastruct in NAL-mode %d failed", inp->of_mode);
610         exit(1);
611     }
612
613     switch(inp->partition_mode)
614     {
615     case PAR_DP_1:
616         currSlice->max_part_nr = 1;
617         break;

```

```

618     case PAR_DP_2:
619         currSlice->max_part_nr = 8;
620         break;
621     case PAR_DP_4:
622         currSlice->max_part_nr = 4;
623         break;
624     default:
625         error("Data Partitioning Mode not supported!");
626         break;
627 }
628
629 currSlice->partArr = (DataPartition *) calloc(currSlice->
max_part_nr, sizeof(DataPartition));
630 if (currSlice->partArr == NULL)
631 {
632     fprintf(stderr, "Memory allocation for Data Partition
datastruct in NAL-mode %d failed", inp->of_mode);
633     exit(1);
634 }
635
636 for (i=0; i<currSlice->max_part_nr; i++) /* loop over all
data partitions */
637 {
638     dataPart = &(currSlice->partArr[i]);
639     dataPart->bitstream = (Bitstream *) calloc(1, sizeof(
Bitstream));
640     if (dataPart->bitstream == NULL)
641     {
642         fprintf(stderr, "Memory allocation for Bitstream
datastruct in NAL-mode %d failed", inp->of_mode);
643         exit(1);
644     }
645     dataPart->bitstream->streamBuffer = (byte *) calloc(
buffer_size, sizeof(byte));
646     if (dataPart->bitstream->streamBuffer == NULL)
647     {
648         fprintf(stderr, "Memory allocation for bitstream
buffer in NAL-mode %d failed", inp->of_mode);
649         exit(1);
650     }
651     /* Initialize storage of bitstream parameters */
652     dataPart->bitstream->stored_bits_to_go = 8;
653     dataPart->bitstream->stored_byte_pos = 0;
654     dataPart->bitstream->stored_byte_buf = 0;
655 }
656 return;
657 default:
658     fprintf(stderr, "Output File Mode %d not supported", inp->
of_mode);
659     exit(1);
660 }
661 }
662 }
663
664 /*****
665 *
666 *   Name :          void free_slice()
667 *
668 *   Description: Memory frees of the Slice structure and of its
dependent
669 *                data structures
670 *
671 *   Input      : Input Parameters struct inp_par *inp,  struct img_par
*img
672 *

```

```

673  ****
674  */
675  void free_slice(struct inp_par *inp, struct img_par *img)
676  {
677      int i;
678      DataPartition *dataPart;
679      Slice *currSlice = img->currentSlice;
680
681      switch(inp->of_mode) /* init depending on NAL mode */
682      {
683          case PAR_OF_26L:
684              /* Current File Format */
685              dataPart = currSlice->partArr; /* only one active data
686              partition */
687              if (dataPart->bitstream->streamBuffer != NULL)
688                  free(dataPart->bitstream->streamBuffer);
689              if (dataPart->bitstream != NULL)
690                  free(dataPart->bitstream);
691              if (currSlice->partArr != NULL)
692                  free(currSlice->partArr);
693              if (inp->symbol_mode == CABAC)
694              {
695                  /* delete all context models */
696                  delete_contexts_MotionInfo(currSlice->mot_ctx);
697                  delete_contexts_TextureInfo(currSlice->tex_ctx);
698              }
699              if (currSlice != NULL)
700                  free(img->currentSlice);
701              break;
702          case PAR_OF_SLICE:
703              /* NAL File Format */
704              for (i=0; i<currSlice->max_part_nr; i++) /* loop over all
705              data partitions */
706              {
707                  dataPart = &(currSlice->partArr[i]);
708                  if (dataPart->bitstream->streamBuffer != NULL)
709                      free(dataPart->bitstream->streamBuffer);
710                  if (dataPart->bitstream != NULL)
711                      free(dataPart->bitstream);
712              }
713              if (currSlice->partArr != NULL)
714                  free(currSlice->partArr);
715              if (currSlice != NULL)
716                  free(img->currentSlice);
717              break;
718          default:
719              fprintf(stderr, "Output File Mode %d not supported", inp->
720              of_mode);
721              exit(1);
722      }
723  }
724
725  /*****
726  *
727  * Name :          void init_stat()
728  *
729  * Description: Initializes the Image structure with appropriate
730  * parameters
731  *
732  * Input      : Input Parameters struct inp_par *inp
733  *
734  * Output     : Statistic Parameters struct stat_par *stat
735  *
736  ****

```

```

733  *****/
734  void init_stat(struct inp_par *inp, struct stat_par *stat)
735  {
736      int i;
737      stat->mode_use_Bframe = (int *)malloc(sizeof(int)*41);
738      stat->bit_use_mode_Bframe = (int *)malloc(sizeof(int)*41);
739      for(i=0; i<41; i++)
740          stat->mode_use_Bframe[i]=stat->bit_use_mode_Bframe[i]=0;
741  }
742
743  /*****
744  *
745  *   Name :          void report()
746  *
747  *   Description: Reports the gathered information to appropriate outputs
748  *
749  *   Input      : struct inp_par *inp,
750  *                struct img_par *img,
751  *                struct stat_par *stat,
752  *                struct stat_par *stat
753  *
754  *   Output     : None
755  *
756  *****/
757  void report(struct inp_par *inp, struct img_par *img, struct stat_par
*stat, struct snr_par *snr)
758  {
759      int bit_use[2][2] ;
760      int i,j;
761      char name[20];
762      int bit_use_Bframe;
763      char timebuf[128];
764
765      bit_use[0][0]=1;
766      bit_use[1][0]=max(1,inp->no_frames-1);
767
768      /* Accumulate bit usage for inter and intra frames */
769      bit_use[0][1]=bit_use[1][1]=0;
770
771      for (i=0; i < 9; i++)
772          bit_use[1][1] += stat->bit_use_mode_inter[i];
773
774      for (j=0; j<2; j++)
775      {
776          bit_use[j][1]+=stat->bit_use_head_mode[j];
777          bit_use[j][1]+=stat->tmp_bit_use_cbp[j];
778          bit_use[j][1]+=stat->bit_use_coeffY[j];
779          bit_use[j][1]+=stat->bit_use_coeffC[j];
780      }
781
782      /* B pictures */
783      if(inp->successive_Bframe!=0 && Bframe_ctr!=0) {
784          bit_use_Bframe=0;
785          for(i=0; i<41; i++)
786              bit_use_Bframe += stat->bit_use_mode_Bframe[i]; //
              fw_predframe_no, blk_size
787          bit_use_Bframe += stat->bit_use_head_mode[2];
788          bit_use_Bframe += stat->tmp_bit_use_cbp[2];
789          bit_use_Bframe += stat->bit_use_coeffY[2];
790          bit_use_Bframe += stat->bit_use_coeffC[2];
791
792          stat->bitrate_P=(stat->bit_ctr_0+stat->bit_ctr_P)*(float) (img
->framerate/(inp->jumpd+1))/inp->no_frames;

```

```

793     stat->bitrate_B=(stat->bit_ctr_B)*(float) (img->framerate/(inp
->jumpd+1))*inp->successive_Bframe/Bframe_ctr;
794 }
795 else {
796     if (inp->no_frames > 1)
797     {
798         stat->bitrate=(bit_use[0][1]+bit_use[1][1])*(float)img->
framerate/(inp->no_frames*(inp->jumpd+1));
799     }
800 }
801
802 fprintf(stdout,
"-----
-----\n");
803 fprintf(stdout, " Freq. for encoded bitstream          : %1.0f\n", (
float)img->framerate/(float) (inp->jumpd+1));
804 if(inp->hadamard)
805     fprintf(stdout, " Hadamard transform                : Used\n");
806 else
807     fprintf(stdout, " Hadamard transform                : Not
used\n");
808
809 if(inp->img_format==0)
810     fprintf(stdout, " Image format                      :
SUB_QCIF\n"); /*GH: other frame sizes added*/
811 else if(inp->img_format==1)
812     fprintf(stdout, " Image format                      : QCIF\n");
813 else if(inp->img_format==2)
814     fprintf(stdout, " Image format                      : CIF\n");
815 else if(inp->img_format==3)
816     fprintf(stdout, " Image format                      : 4CIF\n");
817 else if(inp->img_format==4)
818     fprintf(stdout, " Image format                      : 16CIF\n");
819 else if(inp->img_format==5)
820     fprintf(stdout, " Image format                      : %dx%d\n",
inp->img_width,inp->img_height);
821
822 if(inp->intra_upd)
823     fprintf(stdout, " Error robustness                  : On\n");
824 else if(inp->img_format==1)
825     fprintf(stdout, " Error robustness                  : Off\n");
826 fprintf(stdout, " Search range                        : %d\n",inp
->search_range);
827 fprintf(stdout, " No of ref. frames used in P pred   : %d\n",inp
->no_multpred);
828 if(inp->successive_Bframe != 0)
829     fprintf(stdout, " No of ref. frames used in B pred   : %d\n",
inp->no_multpred);
830 fprintf(stdout, " Total encoding time for the seq.   : %.3f sec
\n",tot_time*0.001);
831
832 /* B pictures */
833 fprintf(stdout, " Sequence type                        : " );
834 if(inp->successive_Bframe==1) fprintf(stdout, " IBPBP (QP: I
%d, P %d, B %d) \n",
835     inp->qp0, inp->qpN, inp->qpB);
836 else if(inp->successive_Bframe==2) fprintf(stdout, " IBBPBBP (QP:
I %d, P %d, B %d) \n",
837     inp->qp0, inp->qpN, inp->qpB);
838 else if(inp->successive_Bframe==0) fprintf(stdout, " IPPP (QP: I
%d, P %d) \n",    inp->qp0, inp->qpN);
839
840 /* DM: report on entropy coding method */
841 if (inp->symbol_mode == UVLC)
842     fprintf(stdout, " Entropy coding method            : UVLC\n");
843 else

```

```

844     fprintf(stdout, " Entropy coding method          : CABAC\n");
845     switch(inp->partition_mode)
846     {
847     case PAR_DP_1:
848         fprintf(stdout, " Data Partitioning Mode      : 1\n");
849         break;
850     case PAR_DP_2:
851         fprintf(stdout, " Data Partitioning Mode      : 2\n");
852         break;
853     case PAR_DP_4:
854         fprintf(stdout, " Data Partitioning Mode      : 4\n");
855         break;
856     default:
857         fprintf(stdout, " Data Partitioning Mode      : not\n");
858         break;
859     }
860
861
862     switch(inp->of_mode)
863     {
864     case PAR_OF_26L:
865         fprintf(stdout, " Output File Format      : H.26L File Format\n");
866         break;
867     case PAR_OF_SLICE:
868         fprintf(stdout, " Output File Format      : Slice\n");
869         break;
870     case PAR_OF_NAL:
871         fprintf(stdout, " Output File Format      : NAL\n");
872         break;
873     default:
874         fprintf(stdout, " Output File Format      : not\n");
875         break;
876     }
877
878
879
880
881     fprintf(stdout, "----- Average data all frames\n");
882     fprintf(stdout, " SNR Y(dB)                : %5.2f\n", snr->snr_ya);
883     fprintf(stdout, " SNR U(dB)                : %5.2f\n", snr->snr_ua);
884     fprintf(stdout, " SNR V(dB)                : %5.2f\n", snr->snr_va);
885
886     if(inp->successive_Bframe != 0) {
887         fprintf(stdout, " Bit rate of Base Layer (kb/s) : %5.2f\n", stat->bitrate_P/1000);
888         fprintf(stdout, " Bit rate of Enhanced Layer (kb/s) : %5.2f\n", stat->bitrate_B/1000);
889         fprintf(stdout, " Total Bit rate (kb/s) : %5.2f\n", stat->bitrate_P/1000+stat->bitrate_B/1000);
890     }
891     else {
892         if (inp->symbol_mode == UVLC)
893             fprintf(stdout, " Total bits : %d\n",

```

```

894         bit_use[0][1]+bit_use[1][1], bit_use[0][1], bit_use[1]
           ][1]);
895     else
896         fprintf(stdout, " Total bits           : %d
           (I %5d, P %5d) \n",
897         stat->bit_ctr_P + stat->bit_ctr_0 , stat->bit_ctr_0, stat
           ->bit_ctr_P);
898         fprintf(stdout, " Bit rate (kb/s)           : %5.2f\n"
           , stat->bitrate/1000);
899     }
900
901     fprintf(stdout,
           "-----
           -----\n");
902     fprintf(stdout, "Exit TML %s encoder ver %s\n", TML, VERSION);
903
904     /* status file */
905     if ((p_stat=fopen("stat.dat","wb"))==0)
906     {
907         printf("Error open file %s \n", p_stat);
908         exit(0);
909     }
910     fprintf(p_stat, "
           ----- \n"
           );
911     fprintf(p_stat, " This file contains statistics for the last
           encoded sequence \n");
912     fprintf(p_stat, "
           ----- \n"
           );
913     fprintf(p_stat, " Sequence           : %s\n", inp->
           infile);
914     fprintf(p_stat, " No.of coded pictures       : %d\n", inp->
           no_frames);
915     fprintf(p_stat, " Freq. for encoded bitstream : %3.0f\n", (float
           )img->framerate/(float) (inp->jumpd+1));
916
917     /* B pictures*/
918     if(inp->successive_Bframe != 0) {
919         fprintf(p_stat, " BaseLayer Bitrate(kb/s)       : %6.2f\n",
           stat->bitrate_P/1000);
920         fprintf(p_stat, " EnhancedLyaer Bitrate(kb/s)    : %6.2f\n",
           stat->bitrate_B/1000);
921     }
922     else
923         fprintf(p_stat, " Bitate(kb/s)           : %6.2f\n",
           stat->bitrate/1000);
924
925     if(inp->hadamard)
926         fprintf(p_stat, " Hadamard transform           : Used\n");
927     else
928         fprintf(p_stat, " Hadamard transform           : Not used\n");
929
930     if(inp->img_format==0)
931         fprintf(p_stat, " Image format           : SUB_QCIF\n");
932         /*GH: other frame sizes added*/
933     else if(inp->img_format==1)
934         fprintf(p_stat, " Image format           : QCIF\n");
935     else if(inp->img_format==2)
936         fprintf(p_stat, " Image format           : CIF\n");
937     else if(inp->img_format==3)
938         fprintf(p_stat, " Image format           : 4CIF\n");
939     else if(inp->img_format==4)
940         fprintf(p_stat, " Image format           : 16CIF\n");
941     else if(inp->img_format==5)
942         fprintf(p_stat, " Image format           : %dx%d\n", inp->

```

```

        img_width,inp->img_height);
942
943     if(inp->intra_upd)
944         fprintf(p_stat," Error robustness           : On\n");
945     else if(inp->img_format==1)
946         fprintf(p_stat," Error robustness           : Off\n");
947
948     fprintf(p_stat, " Search range           : %d\n",inp->
search_range);
949     fprintf(p_stat, " No of frame used in P pred   : %d\n",inp->
no_multpred);
950     if(inp->successive_Bframe != 0)
951         fprintf(p_stat, " No of frame used in B pred   : %d\n",inp->
no_multpred);
952     if (inp->symbol_mode == UVLC)
953         fprintf(p_stat, " Entropy coding method       : UVLC\n");
954     else
955         fprintf(p_stat, " Entropy coding method       : CABAC\n");
956
957     fprintf(p_stat,"
-----|-----|-----|\n");
958     fprintf(p_stat,"      Item           |      Intra       |      All
frames |\n");
959     fprintf(p_stat,"
-----|-----|-----|\n");
960     fprintf(p_stat," SNR Y(dB)           |");
961     fprintf(p_stat," %5.2f           |",snr->snr_y1);
962     fprintf(p_stat," %5.2f           |\n",snr->snr_ya);
963     fprintf(p_stat," SNR U/V (dB)           |");
964     fprintf(p_stat," %5.2f/%5.2f       |",snr->snr_ul,snr->snr_v1);
965     fprintf(p_stat," %5.2f/%5.2f       |\n",snr->snr_ua,snr->snr_va);
966
967     /* QUANT. */
968     fprintf(p_stat," Average quant           |");
969     fprintf(p_stat," %5d           |",absm(inp->qp0));
970     fprintf(p_stat," %5.2f           |\n",(float)stat->quant1/max(1.0,(
float)stat->quant0));
971
972     /* MODE */
973     fprintf(p_stat,"\n -----|-----|\n");
974     fprintf(p_stat,"      Intra           |      Mode used   |\n");
975     fprintf(p_stat," -----|-----|\n");
976
977     fprintf(p_stat," Mode 0  intra old\t| %5d           |\n",stat->
mode_use_intra[0]);
978     for (i=1;i<24;i++)
979         fprintf(p_stat," Mode %d intra new\t| %5d           |\n",i,stat
->mode_use_intra[i]);
980
981     fprintf(p_stat,"\n
-----|-----|-----|\n");
982     fprintf(p_stat,"      Inter           |      Mode used   | Vector bit
use|\n");
983     fprintf(p_stat,"
-----|-----|-----|");
984
985     fprintf(p_stat,"\n Mode 0  (copy) \t| %5d           |
%5.0f           |",stat->mode_use_inter[0],(float)stat->
bit_use_mode_inter[0]/(float)bit_use[1][0]);
986     fprintf(p_stat,"\n Mode 1  (16x16) \t| %5d           |
%5.0f           |",stat->mode_use_inter[1],(float)stat->
bit_use_mode_inter[1]/(float)bit_use[1][0]);
987     fprintf(p_stat,"\n Mode 2  (16x8) \t| %5d           |
%5.0f           |",stat->mode_use_inter[2],(float)stat->
bit_use_mode_inter[2]/(float)bit_use[1][0]);
988     fprintf(p_stat,"\n Mode 3  (8x16) \t| %5d           |

```

```

    %5.0f          |",stat->mode_use_inter[3],(float)stat->
    bit_use_mode_inter[3]/(float)bit_use[1][0]);
989    fprintf(p_stat,"\n Mode 4   (8x8)   \t| %5d          |
    %5.0f          |",stat->mode_use_inter[4],(float)stat->
    bit_use_mode_inter[4]/(float)bit_use[1][0]);
990    fprintf(p_stat,"\n Mode 5   (8x4)   \t| %5d          |
    %5.0f          |",stat->mode_use_inter[5],(float)stat->
    bit_use_mode_inter[5]/(float)bit_use[1][0]);
991    fprintf(p_stat,"\n Mode 6   (4x8)   \t| %5d          |
    %5.0f          |",stat->mode_use_inter[6],(float)stat->
    bit_use_mode_inter[6]/(float)bit_use[1][0]);
992    fprintf(p_stat,"\n Mode 7   (4x4)   \t| %5d          |
    %5.0f          |",stat->mode_use_inter[7],(float)stat->
    bit_use_mode_inter[7]/(float)bit_use[1][0]);
993    fprintf(p_stat,"\n Mode 8 intra old\t| %5d          |
    -----|",stat->mode_use_inter[8],(float)stat->
    bit_use_mode_inter[8]/(float)bit_use[1][0]);
994    for (i=9;i<33;i++)
995        fprintf(p_stat,"\n Mode %d intr.new \t| %5d          |",i,stat
        ->mode_use_inter[i]);

996
997    /* B pictures */
998    if(inp->successive_Bframe!=0 && Bframe_ctr!=0) {
999        fprintf(p_stat,"\n\n -----|-----|\n");
1000        fprintf(p_stat,"      B frame          |      Mode used      |\n");
1001        fprintf(p_stat," -----|-----|");
1002
1003        for(i=0; i<16; i++)
1004            fprintf(p_stat,"\n Mode %d      \t| %5d          |", i, stat->
            mode_use_Bframe[i]);
1005        fprintf(p_stat,"\n Mode %d intra old\t| %5d          |", 16, stat->
            mode_use_Bframe[16]);
1006        for (i=17; i<41; i++)
1007            fprintf(p_stat,"\n Mode %d intr.new \t| %5d          |",i, stat
            ->mode_use_Bframe[i]);
1008    }
1009
1010    fprintf(p_stat,"\n\n
    -----|-----|-----
    |\n");
1011    fprintf(p_stat,"  Bit usage:          |      Intra      |
    Inter      |      B frame      |\n");
1012    fprintf(p_stat,"
    -----|-----|-----|-----
    |\n");

1013
1014    fprintf(p_stat," Header+modeinfo      |");
1015    fprintf(p_stat," %7d          |",stat->bit_use_head_mode[0]);
1016    fprintf(p_stat," %7d          |",stat->bit_use_head_mode[1]/bit_use[1
    ][0]);
1017    if(inp->successive_Bframe!=0 && Bframe_ctr!=0)
1018        fprintf(p_stat," %7d          |",stat->bit_use_head_mode[2]/
            Bframe_ctr);
1019    else fprintf(p_stat," %7d          |", 0);
1020    fprintf(p_stat,"\n");
1021
1022    fprintf(p_stat," CBP Y/C          |");
1023    for (j=0; j < 2; j++)
1024    {
1025        fprintf(p_stat," %7.2f          |", (float)stat->tmp_bit_use_cbp[
            j]/bit_use[j][0]);
1026    }
1027    if(inp->successive_Bframe!=0 && Bframe_ctr!=0)
1028        fprintf(p_stat," %7.2f          |", (float)stat->tmp_bit_use_cbp[
            2]/Bframe_ctr);
1029    else fprintf(p_stat," %7.2f          |", 0.);

```

```

1030     fprintf(p_stat, "\n");
1031
1032     if(inp->successive_Bframe!=0 && Bframe_ctr!=0)
1033         fprintf(p_stat, " Coeffs. Y          | %7.2f          |
1034         %7.2f          | %7.2f          |\n",
1035         (float)stat->bit_use_coeffY[0]/bit_use[0][0], (float)stat
1036         ->bit_use_coeffY[1]/bit_use[1][0], (float)stat->
1037         bit_use_coeffY[2]/Bframe_ctr);
1038
1039     else
1040         fprintf(p_stat, " Coeffs. Y          | %7.2f          |
1041         %7.2f          | %7.2f          |\n",
1042         (float)stat->bit_use_coeffY[0]/bit_use[0][0], (float)stat
1043         ->bit_use_coeffY[1]/bit_use[1][0], 0.);
1044
1045     if(inp->successive_Bframe!=0 && Bframe_ctr!=0)
1046         fprintf(p_stat, " Coeffs. C          | %7.2f          |
1047         %7.2f          | %7.2f          |\n",
1048         (float)stat->bit_use_coeffC[0]/bit_use[0][0], (float)stat
1049         ->bit_use_coeffC[1]/bit_use[1][0], (float)stat->
1050         bit_use_coeffC[2]/Bframe_ctr);
1051
1052     else
1053         fprintf(p_stat, " Coeffs. C          | %7.2f          |
1054         %7.2f          | %7.2f          |\n",
1055         (float)stat->bit_use_coeffC[0]/bit_use[0][0], (float)stat
1056         ->bit_use_coeffC[1]/bit_use[1][0], 0.);
1057
1058     fprintf(p_stat, "
1059     -----|-----|-----|-----
1060     |\n");
1061
1062     fprintf(p_stat, " average bits/frame |");
1063     for (i=0; i < 2; i++)
1064     {
1065         fprintf(p_stat, " %7d          |", bit_use[i][1]/bit_use[i][0]);
1066     }
1067     if(inp->successive_Bframe!=0 && Bframe_ctr!=0)
1068         fprintf(p_stat, " %7d          |", bit_use_Bframe/Bframe_ctr);
1069     else fprintf(p_stat, " %7d          |", 0);
1070
1071     fprintf(p_stat, "\n");
1072     fprintf(p_stat, "
1073     -----|-----|-----|-----
1074     |\n");
1075
1076     /* write to log file */
1077     if (fopen("log.dat", "r")==0) /* check if
1078     file exist */
1079     {
1080         if ((p_log=fopen("log.dat", "a"))==0) /* append new
1081         statistic at the end */
1082         {
1083             printf("Error open file %s \n", p_log);
1084             exit(0);
1085         }
1086         else /* Create
1087         header for new log file */
1088         {
1089             fprintf(p_log, "
1090             -----
1091             -----
1092             ----- \n");
1093             fprintf(p_log, "|          Encoder statistics. This file
1094             is generated during first encoding session, new sessions
1095             will be
1096             appended
1097             |\n");

```

```

1072         fprintf(p_log,"
-----
-----
----- \n");

1073         fprintf(p_log,"| Date | Time | Sequence
| #Img|Quant1|QuantN|Format|Hadamard|Search r|#Ref |Freq
|Intra upd|SNRY 1|SNRU 1|SNRV 1|SNRY N|SNRU N|SNRV
N|#Bitr P|#Bitr B|\n");

1074         fprintf(p_log,"
-----
-----
----- \n");

1075     }
1076 }
1077 else
1078     p_log=fopen("log.dat","a"); /* File
    exist,just open for appending */

1079
1080 #ifdef WIN32
1081     _strdate( timebuf );
1082     fprintf(p_log,"| %1.5s |",timebuf );
1083
1084     _strtime( timebuf);
1085     fprintf(p_log," % 1.5s |",timebuf);
1086 #else
1087     now = time ((time_t *) NULL); /* Get the system time and put it
    into 'now' as 'calender time' */
1088     time (&now);
1089     l_time = localtime (&now);
1090     strftime (string, sizeof string, "%d-%b-%y", l_time);
1091     fprintf(p_log,"| %1.5s |",string );
1092
1093     strftime (string, sizeof string, "%H:%M:%S", l_time);
1094     fprintf(p_log,"| %1.5s |",string );
1095 #endif
1096
1097     for (i=0;i<20;i++)
1098         name[i]=inp->infile[i+max(0,strlen(inp->infile)-20)]; /*
    write last part of path, max 20 chars */
1099     fprintf(p_log,"%20.20s|",name);
1100
1101     fprintf(p_log,"%3d |",inp->no_frames);
1102     fprintf(p_log," %2d |",inp->qp0);
1103     fprintf(p_log," %2d |",inp->qpN);
1104
1105
1106     if(inp->img_format==0) /*GH: other frame sizes added*/
1107         fprintf(p_log,"S_QCIF|");
1108     else if(inp->img_format==1)
1109         fprintf(p_log," QCIF |");
1110     else if(inp->img_format==2)
1111         fprintf(p_log," CIF |");
1112     else if(inp->img_format==3)
1113         fprintf(p_log," 4CIF |");
1114     else if(inp->img_format==4)
1115         fprintf(p_log," 16CIF|");
1116     else if(inp->img_format==5)
1117         fprintf(p_log,"%dx%d|",inp->img_width,inp->img_height);
1118
1119
1120     if (inp->hadamard==1)
1121         fprintf(p_log," ON |");
1122     else
1123         fprintf(p_log," OFF |");
1124
1125     fprintf(p_log," %2d |",inp->search_range );

```

```

1126
1127     fprintf(p_log, " %2d  |", inp->no_multpred);
1128
1129     fprintf(p_log, " %2d  |", img->framerate/(inp->jumpd+1));
1130
1131     if (inp->intra_upd==1)
1132         fprintf(p_log, "    ON    |");
1133     else
1134         fprintf(p_log, "    OFF   |");
1135
1136     fprintf(p_log, "%5.3f|", snr->snr_y1);
1137     fprintf(p_log, "%5.3f|", snr->snr_ul);
1138     fprintf(p_log, "%5.3f|", snr->snr_v1);
1139     fprintf(p_log, "%5.3f|", snr->snr_ya);
1140     fprintf(p_log, "%5.3f|", snr->snr_ua);
1141     fprintf(p_log, "%5.3f|", snr->snr_va);
1142     if(inp->successive_Bframe != 0)
1143     {
1144         fprintf(p_log, "%7.0f|", stat->bitrate_P);
1145         fprintf(p_log, "%7.0f|\n", stat->bitrate_B);
1146     }
1147     else
1148     {
1149         fprintf(p_log, "%7.0f|", stat->bitrate);
1150         fprintf(p_log, "%7.0f|\n", 0.0);
1151     }
1152
1153     fclose(p_log);
1154
1155     p_log=fopen("data.txt", "a");
1156
1157     if(inp->successive_Bframe != 0) // B picture used
1158     {
1159         fprintf(p_log, "%3d %2d %2d %2.2f %2.2f %2.2f %5d "
1160             "%2.2f %2.2f %2.2f %5d "
1161             "%2.2f %2.2f %2.2f %5d %5d %.3f\n",
1162             inp->no_frames, inp->qp0, inp->qpN,
1163             snr->snr_y1,
1164             snr->snr_ul,
1165             snr->snr_v1,
1166             stat->bit_ctr_0,
1167             0.0,
1168             0.0,
1169             0.0,
1170             0,
1171             snr->snr_ya,
1172             snr->snr_ua,
1173             snr->snr_va,
1174             (stat->bit_ctr_0+stat->bit_ctr)/inp->no_frames,
1175             stat->bit_ctr_B/Bframe_ctr,
1176             (double)0.001*tot_time/(inp->no_frames+Bframe_ctr));
1177     }
1178     else
1179     {
1180         fprintf(p_log, "%3d %2d %2d %2.2f %2.2f %2.2f %5d "
1181             "%2.2f %2.2f %2.2f %5d "
1182             "%2.2f %2.2f %2.2f %5d %5d %.3f\n",
1183             inp->no_frames, inp->qp0, inp->qpN,
1184             snr->snr_y1,
1185             snr->snr_ul,
1186             snr->snr_v1,
1187             stat->bit_ctr_0,
1188             0.0,
1189             0.0,
1190             0.0,
1191             0,

```

```

1192         snr->snr_ya,
1193         snr->snr_ua,
1194         snr->snr_va,
1195         (stat->bit_ctr_0+stat->bit_ctr)/inp->no_frames,
1196         0,
1197         (double)0.001*tot_time/inp->no_frames);
1198     }
1199
1200     fclose(p_log);
1201
1202     /* B pictures */
1203     free(stat->mode_use_Bframe);
1204     free(stat->bit_use_mode_Bframe);
1205
1206 }
1207
1208 /*****
1209  *
1210  * Name :      init()
1211  *
1212  * Description: Some matrices are initilized here.
1213  *
1214  * Input      : none
1215  *
1216  * Output     : none
1217  *
1218  *****/
1219 void init(struct img_par *img)
1220 {
1221     int i,j,k,l,i2,l2,ii,ind;
1222
1223     k=1;
1224     for (l=1; l < 41; l++)
1225     {
1226         l2=l+1;
1227
1228         for (i=-l+1; i < l; i++)
1229         {
1230             for (j=-1; j < l+1; j += l2)
1231             {
1232                 /* img->spiral_search contains the vector positions
1233                  organized to perform a spiral search */
1234                 img->spiral_search[0][k]= i;
1235                 img->spiral_search[1][k]= j;
1236                 k++;
1237             }
1238
1239             for (j=-1; j <= l; j++)
1240             {
1241                 for (i=-1;i <= l; i += l2)
1242                 {
1243                     img->spiral_search[0][k]= i;
1244                     img->spiral_search[1][k]= j;
1245                     k++;
1246                 }
1247             }
1248         }
1249
1250         /* img->lpfilter is the table used for loop and post filter. */
1251
1252         for (k=0; k < 6; k++)
1253         {
1254             for (i=1; i <= 100; i++)

```

```

1255     {
1256         ii=min(k,i);
1257         img->lpfilter[k][i+300]= ii;
1258         img->lpfilter[k][300-i]=-ii;
1259     }
1260 }
1261 /* img->mv_bituse[] is the number of bits used for motion vectors.
1262 It is used to add a portion to the SAD depending on bit usage in
1263 the motion search
1264 */
1265 img->mv_bituse[0]=1;
1266 ind=0;
1267 for (i=0; i < 9; i++)
1268 {
1269     ii= 2*i + 3;
1270     for (j=1; j <= (int)pow(2,i); j++)
1271     {
1272         ind++;
1273         img->mv_bituse[ind]=ii;
1274     }
1275 }
1276
1277 /* quad(0:255) SNR quad array */
1278 for (i=0; i < 256; ++i) /* fix from TML1 / TML2 sw, truncation
1279 removed */
1280 {
1281     i2=i*i;
1282     img->quad[i]=i2;
1283 }
1284 /* B pictures : img->blk_bitsue[] is used when getting
1285 bidirection SAD */
1286 for (i=0; i < 7; i++)
1287 {
1288     if(i==0) img->blk_bituse[i]=1;
1289     else if(i==1 || i==2) img->blk_bituse[i]=3;
1290     else img->blk_bituse[i]=5;
1291 }
1292
1293 /*****
1294 *
1295 * Name :      information_init()
1296 *
1297 * Description: Prints the header of the protocol
1298 *
1299 * Input      : struct inp_par *inp, char *config_filename
1300 *
1301 * Output     : none
1302 *
1303 *****/
1304 void information_init(struct inp_par *inp, char *config_filename)
1305 {
1306     printf(
1307         "-----\n");
1308     printf(" Encoder config file          : %s \n",
1309         config_filename);
1310     printf(
1311         "-----\n");
1312     printf(" Input YUV file                  : %s \n",inp->infile);
1313     printf(" Output H.26L bitstream          : %s \n",inp->outfile);

```

```

1311     if (inp->write_dec)
1312         printf(" Output YUV file(debug)           : test.dec \n");
1313     printf(" Output log file                       : log.dat \n");
1314     printf(" Output statistics file                     : stat.dat \n");
1315     printf(
1316         "-----\n");
1317
1318     printf(" Frame  Bit/pic   QP   SnrY   SnrU   SnrV   Time(ms) \n"
1319 );
1320 }
1321
1322 /*****
1323 *
1324 * Name :      int get_mem4global_buffers(struct inp_par *inp,
1325 struct img_par *img)
1326 *
1327 * Description: Dynamic memory allocation of frame size related
1328 global buffers
1329 *
1330 buffers are defined in global.h, allocated memory
1331 must be freed in
1332 void free_mem4global_buffers()
1333 *
1334 * Input      : Input Parameters struct inp_par *inp, Image
1335 Parameters struct img_par *img
1336 *
1337 * Output     : Number of allocated bytes
1338 *
1339 *****/
1340 int get_mem4global_buffers(struct inp_par *inp, struct img_par *img)
1341 {
1342     int j, memory_size=0;
1343
1344     /* allocate memory for encoding frame buffers: imgY, imgUV*/
1345     /* byte imgY[288][352];      */
1346     /* byte imgUV[2][144][176];  */
1347     memory_size += get_mem2D(&imgY, img->height, img->width);
1348     memory_size += get_mem3D(&imgUV, 2, img->height_cr, img->width_cr);
1349
1350     /* allocate memory for reference frame buffers: imgY_org,
1351 imgUV_org*/
1352     /* byte imgY_org[288][352];    */
1353     /* byte imgUV_org[2][144][176]; */
1354     memory_size += get_mem2D(&imgY_org, img->height, img->width);
1355     memory_size += get_mem3D(&imgUV_org, 2, img->height_cr, img->
1356 width_cr);
1357
1358     /* allocate memory for post filter frame buffers: imgY_pf,
1359 imgUV_pf*/
1360     /* byte imgY_pf[288][352];      */
1361     /* byte imgUV_pf[2][144][176];  */
1362     memory_size += get_mem2D(&imgY_pf, img->height, img->width);
1363     memory_size += get_mem3D(&imgUV_pf, 2, img->height_cr, img->
1364 width_cr);
1365
1366     /* allocate memory for B frame coding: nextP_imgY, nextP_imgUV*/
1367     /* byte nextP_imgY[288][352];    */
1368     /* byte nextP_imgUV[2][144][176]; */
1369     memory_size += get_mem2D(&nextP_imgY, img->height, img->width);
1370     memory_size += get_mem3D(&nextP_imgUV, 2, img->height_cr, img->
1371 width_cr);
1372

```

```

1363     /* allocate memory for multiple ref. frame buffers: mref, mcref*/
1364     /*byte mref[MAX_MULT_PRED][1152][1408]; */ /* 1/4 pix luma */
1365     /*byte mcef[MAX_MULT_PRED][2][352][288]; */ /* pix chroma */
1366     /* rows and cols for croma component mcef[ref][croma][4x][4y] are
       switched */
1367     /* compared to luma mref[ref][4y][4x] for whatever reason */
1368     /* number of reference frames increased by one for next P-frame*/
1369     memory_size += get_mem3D(&mref, inp->no_multpred+1, img->height*4,
       img->width*4);

1370
1371     if((mcef = (byte****)calloc(inp->no_multpred+1, sizeof(byte***)))
       == NULL) no_mem_exit(1);
1372     for(j=0; j<inp->no_multpred+1; j++)
1373     {
1374         memory_size += get_mem3D(&mcef[j]), 2, img->width_cr*2, img->
       height_cr*2);
1375     }
1376
1377
1378     /* allocate memory for B-frame coding (last upsampled P-frame):
       mref_P, mcref_P*/
1379     /* is currently also used in oneforthpix_1() and _2() for coding
       without B-frames*/
1380     /*byte mref[1152][1408]; */ /* 1/4 pix luma */
1381     /*byte mcef[2][352][288]; */ /* pix chroma */
1382     memory_size += get_mem2D(&mref_P, img->height*4, img->width*4);
1383     memory_size += get_mem3D(&mcef_P, 2, img->width_cr*2, img->
       height_cr*2);

1384
1385     if(inp->successive_Bframe!=0)
1386     {
1387         /* allocate memory for temp B-frame motion vector buffer:
       tmp_fwMV, tmp_bwMV, dfMV, dbMV*/
1388         /* int ...MV[2][72][92]; ([2][72][88] should be enough) */
1389         memory_size += get_mem3Dint(&tmp_fwMV, 2, img->height/BLOCK_SIZE,
       img->width/BLOCK_SIZE+4);
1390         memory_size += get_mem3Dint(&tmp_bwMV, 2, img->height/BLOCK_SIZE,
       img->width/BLOCK_SIZE+4);
1391         memory_size += get_mem3Dint(&dfMV, 2, img->height/BLOCK_SIZE,
       img->width/BLOCK_SIZE+4);
1392         memory_size += get_mem3Dint(&dbMV, 2, img->height/BLOCK_SIZE,
       img->width/BLOCK_SIZE+4);

1393
1394         /* allocate memory for temp B-frame motion vector buffer:
       fw_refFrArr, bw_refFrArr*/
1395         /* int ...refFrArr[72][88]; */
1396         memory_size += get_mem2Dint(&fw_refFrArr, img->height/BLOCK_SIZE,
       img->width/BLOCK_SIZE);
1397         memory_size += get_mem2Dint(&bw_refFrArr, img->height/BLOCK_SIZE,
       img->width/BLOCK_SIZE);
1398     }
1399
1400     /* allocate memory for temp P and B-frame motion vector buffer:
       tmp_mv, temp_mv_block*/
1401     /* int tmp_mv[2][72][92]; ([2][72][88] should be enough) */
1402     memory_size += get_mem3Dint(&tmp_mv, 2, img->height/BLOCK_SIZE,
       img->width/BLOCK_SIZE+4);

1403
1404     /* allocate memory for temp quarter pel luma frame buffer:
       img4Y_tmp*/
1405     /* int img4Y_tmp[576][704]; (previously int imgY_tmp in
       global.h) */
1406     memory_size += get_mem2Dint(&img4Y_tmp, img->height*2, img->width*
       2);

1407
1408     /* allocate memory for tmp loop filter frame buffers: imgY_tmp,

```

```

1409     imgUV_tmp*/
1410     /* byte imgY_tmp[288][352]; */
1411     memory_size += get_mem2D(&imgY_tmp, img->height, img->width);
1412     memory_size += get_mem3D(&imgUV_tmp, 2, img->height_cr, img->
width_cr);
1413
1414     /* allocate memory for reference frames of each block: refFrArr*/
1415     /* int refFrArr[72][88]; */
1416     memory_size += get_mem2Dint(&refFrArr, img->height/BLOCK_SIZE, img
->width/BLOCK_SIZE);
1417
1418     /* allocate memory for filter strength for 4x4 lums and croma
blocks:loopb, loopc*/
1419     /*byte loopb[91][75]; i.e.[x][y]*/
1420     /*byte loopc[47][39]; */
1421     memory_size += get_mem2D(&loopb, img->width/BLOCK_SIZE+3, img->
height/BLOCK_SIZE+3);
1422     memory_size += get_mem2D(&loopc, img->width_cr/BLOCK_SIZE+3, img->
height_cr/BLOCK_SIZE+3);
1423
1424     return (memory_size);
1425 }
1426
1427 /*****
1428 *
1429 * Name : void free_mem4global_buffers(struct inp_par *inp,
struct img_par *img)
1430 *
1431 * Description: Free allocated memory of frame size related global
buffers
1432 *
1433 * buffers are defined in global.h, allocated memory is
allocated in
1434 *
1435 * int get_mem4global_buffers()
1436 *
1437 * Input : Input Parameters struct inp_par *inp, Image
Parameters struct img_par *img
1438 *
1439 * Output : none
1440 *
1441 *****/
1442 void free_mem4global_buffers(struct inp_par *inp, struct img_par *img)
1443 {
1444     int i,j;
1445
1446     for(i=0;i<img->height;i++)
1447     {
1448         free(imgY[i]);
1449     }
1450     free(imgY);
1451
1452     for(i=0;i<img->height_cr;i++)
1453     {
1454         free(imgUV[0][i]);
1455         free(imgUV[1][i]);
1456     }
1457     free(imgUV[0]);
1458     free(imgUV[1]);
1459     free(imgUV);
1460
1461     /* free ref frame buffers*/
1462     for(i=0;i<img->height;i++)
1463     {
1464         free(imgY_org[i]);
1465     }

```

```
1463     free(imgY_org);
1464
1465     for(i=0;i<img->height_cr;i++)
1466     {
1467         free(imgUV_org[0][i]);
1468         free(imgUV_org[1][i]);
1469     }
1470     free(imgUV_org[0]);
1471     free(imgUV_org[1]);
1472     free(imgUV_org);
1473
1474     /* free post filtering frame buffers*/
1475     for(i=0;i<img->height;i++)
1476     {
1477         free(imgY_pf[i]);
1478     }
1479     free(imgY_pf);
1480
1481     for(i=0;i<img->height_cr;i++)
1482     {
1483         free(imgUV_pf[0][i]);
1484         free(imgUV_pf[1][i]);
1485     }
1486     free(imgUV_pf[0]);
1487     free(imgUV_pf[1]);
1488     free(imgUV_pf);
1489
1490     /* free next frame buffers (for B frames)*/
1491     for(i=0;i<img->height;i++)
1492     {
1493         free(nextP_imgY[i]);
1494     }
1495     free(nextP_imgY);
1496
1497     for(i=0;i<img->height_cr;i++)
1498     {
1499         free(nextP_imgUV[0][i]);
1500         free(nextP_imgUV[1][i]);
1501     }
1502     free(nextP_imgUV[0]);
1503     free(nextP_imgUV[1]);
1504     free(nextP_imgUV);
1505
1506
1507     /* free multiple ref frame buffers*/
1508     /* number of reference frames increased by one for next P-frame*/
1509     for(j=0;j<inp->no_multpred+1;j++)
1510     {
1511         for(i=0;i<img->height*4;i++)
1512         {
1513             free(mref[j][i]);
1514         }
1515         free(mref[j]);
1516
1517         for(i=0;i<img->width_cr*2;i++)
1518         {
1519             free(mcef[j][0][i]);
1520             free(mcef[j][1][i]);
1521         }
1522         free(mcef[j][0]);
1523         free(mcef[j][1]);
1524         free(mcef[j]);
1525     }
1526     free(mref);
1527     free(mcef);
1528
```

```
1529     for(i=0;i<img->height*4;i++)
1530     {
1531         free(mref_P[i]);
1532     }
1533     free(mref_P);
1534
1535     for(i=0;i<img->width_cr*2;i++)
1536     {
1537         free(mcef_P[0][i]);
1538         free(mcef_P[1][i]);
1539     }
1540     free(mcef_P[0]);
1541     free(mcef_P[1]);
1542     free(mcef_P);
1543
1544     if(inp->successive_Bframe!=0)
1545     {
1546         /* free last P-frame buffers for B-frame coding*/
1547         for(j=0;j<2;j++)
1548         {
1549             for(i=0;i<img->height/BLOCK_SIZE;i++)
1550             {
1551                 free(tmp_fwMV[j][i]);
1552                 free(tmp_bwMV[j][i]);
1553                 free(dfMV[j][i]);
1554                 free(dbMV[j][i]);
1555             }
1556             free(tmp_fwMV[j]);
1557             free(tmp_bwMV[j]);
1558             free(dfMV[j]);
1559             free(dbMV[j]);
1560         }
1561         free(tmp_fwMV);
1562         free(tmp_bwMV);
1563         free(dfMV);
1564         free(dbMV);
1565
1566
1567         for(i=0;i<img->height/BLOCK_SIZE;i++)
1568         {
1569             free(fw_refFrArr[i]);
1570             free(bw_refFrArr[i]);
1571         }
1572         free(fw_refFrArr);
1573         free(bw_refFrArr);
1574     } /* end if B frame*/
1575
1576
1577     for(j=0;j<2;j++)
1578     {
1579         for(i=0;i<img->height/BLOCK_SIZE;i++)
1580         {
1581             free(tmp_mv[j][i]);
1582         }
1583         free(tmp_mv[j]);
1584     }
1585     free(tmp_mv);
1586
1587     /* free temp quarter pel frame buffer*/
1588     for(i=0;i<img->height*2;i++)
1589     {
1590         free(img4Y_tmp[i]);
1591     }
1592     free(img4Y_tmp);
1593
1594
```

```
1595     /* free temp loop filter frame buffer*/
1596     for(i=0;i<img->height;i++)
1597     {
1598         free(imgY_tmp[i]);
1599     }
1600     free(imgY_tmp);
1601
1602     for(i=0;i<img->height_cr;i++)
1603     {
1604         free(imgUV_tmp[0][i]);
1605         free(imgUV_tmp[1][i]);
1606     }
1607     free(imgUV_tmp[0]);
1608     free(imgUV_tmp[1]);
1609     free(imgUV_tmp);
1610
1611     /* free ref frame buffer for blocks*/
1612     for(i=0;i<img->height/BLOCK_SIZE;i++)
1613     {
1614         free(refFrArr[i]);
1615     }
1616     free(refFrArr);
1617
1618     /* free loop filter strength buffer for 4x4 blocks*/
1619     for(i=0;i<img->width/BLOCK_SIZE+3;i++)
1620     {
1621         free(loopb[i]);
1622     }
1623     free(loopb);
1624
1625     for(i=0;i<img->width_cr/BLOCK_SIZE+3;i++)
1626     {
1627         free(loopc[i]);
1628     }
1629     free(loopc);
1630
1631     /* free mem, allocated in init_img()*/
1632     /* free intra pred mode buffer for blocks*/
1633     for(i=0;i<img->width/BLOCK_SIZE+2;i++)
1634     {
1635         free(img->ipredmode[i]);
1636     }
1637     free(img->ipredmode);
1638
1639     free(img->mb_data);
1640
1641     free(img->slice_numbers);
1642 }
1643
1644 /*****
1645 *
1646 *   Name :          int get_mem2D(byte ***array2D, int rows, int columns)
1647 *
1648 *   Description: Allocate 2D memory array -> unsigned char
1649 *   array2D[rows][columns]
1650 *
1651 *   Input      :
1652 *
1653 *   Output     : memory size in bytes
1654 *
1655 *****/
1656 int get_mem2D(byte ***array2D, int rows, int columns)
1657 {
1658     int i;
```

```

1658
1659     if((*array2D = (byte**)calloc(rows,sizeof(byte*))) == NULL)
        no_mem_exit(1);
1660     for(i=0;i<rows;i++)
1661     {
1662         if((*array2D)[i] = (byte*)calloc(columns,sizeof(byte))) == NULL)
            no_mem_exit(1);
1663     }
1664     return rows*columns;
1665 }
1666
1667 /*****
1668  *
1669  *   Name :          int get_mem2Dint(int ***array2D, int rows, int columns)
1670  *
1671  *   Description: Allocate 2D memory array -> int
        array2D[rows][columns]
1672  *
1673  *   Input      :
1674  *
1675  *   Output     : memory size in bytes
1676  *
1677  *****/
1678 int get_mem2Dint(int ***array2D, int rows, int columns)
1679 {
1680     int i;
1681
1682     if((*array2D = (int**)calloc(rows,sizeof(int*))) == NULL)
        no_mem_exit(1);
1683     for(i=0;i<rows;i++)
1684     {
1685         if((*array2D)[i] = (int*)calloc(columns,sizeof(int))) == NULL)
            no_mem_exit(1);
1686     }
1687     return rows*columns*sizeof(int);
1688 }
1689
1690 /*****
1691  *
1692  *   Name :          int get_mem3D(byte ****array3D, int frames, int rows,
        int columns)
1693  *
1694  *   Description: Allocate 3D memory array -> unsigned char
        array3D[frames][rows][columns]
1695  *
1696  *   Input      :
1697  *
1698  *   Output     : memory size in bytes
1699  *
1700  *****/
1701 int get_mem3D(byte ****array3D, int frames, int rows, int columns)
1702 {
1703     int i, j;
1704
1705     if((*array3D = (byte***)calloc(frames,sizeof(byte**))) == NULL)
        no_mem_exit(1);
1706
1707     for(j=0;j<frames;j++)
1708     {
1709         if((*array3D)[j] = (byte**)calloc(rows,sizeof(byte*))) == NULL)
            no_mem_exit(1);
1710         for(i=0;i<rows;i++)

```

```

1711     {
1712         if(((array3D)[j][i] = (byte*) calloc(columns, sizeof(byte))) ==
            NULL) no_mem_exit(1);
1713     }
1714 }
1715
1716 return frames*rows*columns;
1717 }
1718 /*****
1719 *
1720 *   Name :      int get_mem3Dint(int ***array3D, int frames, int
            rows, int columns)
1721 *
1722 *   Description: Allocate 3D memory array -> int
            array3D[frames][rows][columns]
1723 *
1724 *   Input      :
1725 *
1726 *   Output     : memory size in bytes
1727 *
1728 *****/
1729 int get_mem3Dint(int ***array3D, int frames, int rows, int columns)
1730 {
1731     int i, j;
1732
1733     if(((array3D) = (int***) calloc(frames, sizeof(int**))) == NULL)
        no_mem_exit(1);
1734
1735     for(j=0; j<frames; j++)
1736     {
1737         if(((array3D)[j] = (int**) calloc(rows, sizeof(int*))) == NULL)
            no_mem_exit(1);
1738         for(i=0; i<rows; i++)
1739         {
1740             if(((array3D)[j][i] = (int*) calloc(columns, sizeof(int))) ==
                NULL) no_mem_exit(1);
1741         }
1742     }
1743
1744     return frames*rows*columns*sizeof(int);
1745 }
1746 /*****
1747 *
1748 *   Name :      void no_mem_exit(int code)
1749 *
1750 *   Description: Exit program if there is not enough memory for frame
            buffers
1751 *
1752 *   Input      : code for exit()
1753 *
1754 *   Output     : none
1755 *
1756 *****/
1757 void no_mem_exit(int code)
1758 {
1759     printf("Could not allocate frame buffer memory!\n");
1760     exit (code);
1761 }
1762

```

```

1  //
   *****
   *****
2  //
   *****
   *****
3  // Macroblock.c  Process one macroblock
4  //
5  // Main contributors (see contributors.h for copyright, address and
   affiliation details)
6  //
7  // Inge Lille-Langøy          <inge.lille-langoy@telenor.com>
8  // Rickard Sjoberg           <rickard.sjoberg@era.ericsson.se>
9  // Jani Lainema              <jani.lainema@nokia.com>
10 // Sebastian Purreiter       <sebastian.purreiter@mch.siemens.de>
11 // Detlev Marpe              <marpe@hhi.de>
12 //
   *****
   *****
13 //
   *****
   *****
14 #include "contributors.h"
15
16 #include <math.h>
17 #include <stdlib.h>
18
19 #include "global.h"
20 #include "elements.h"
21 #include "macroblock.h"
22
23
24 /*****
   **
25  *
26  *   Name :          proceed2nextMacroblock()
27  *
28  *   Description: updates the coordinates and statistics parameter for
   the
29  *               next macroblock
30  *
31  *****/
32 void proceed2nextMacroblock(struct img_par *img, struct inp_par *inp,
   struct stat_par *stat)
33 {
34
35     int use_bitstream_backing = (inp->slice_mode == FIXED_RATE || inp
   ->slice_mode == CALLBACK);
36     const int number_mb_per_row = img->width / MB_BLOCK_SIZE ;
37     Macroblock *currMB = &img->mb_data[img->current_mb_nr];
38
39     #if TRACE
40     int i;
41     if(use_bitstream_backing)
42         fprintf(p_trace, "\n***** Pic: %i (I/P) MB: %i Slice:
   %i *****\n\n", frame_no, img->current_mb_nr, img->
   current_slice_nr);
43     /* Write out the tracestring for each symbol */
44     for (i=0; i<currMB->currSEnr; i++)
45         trace2out(&(img->MB_SyntaxElements[i]));
46     #endif
47
48     /* Update the statistics */
49     stat->bit_use_head_mode[img->type]      += currMB->bitcounter[
   BITS_HEADER_MB];

```

```

50     stat->bit_use_coeffY[img->type]           += currMB->bitcounter[
        BITS_COEFF_Y_MB] ;
51     stat->bit_use_mode_inter[img->mb_mode] += currMB->bitcounter[
        BITS_INTER_MB];
52     stat->tmp_bit_use_cbp[img->type]           += currMB->bitcounter[
        BITS_CBP_MB];
53     stat->bit_use_coeffC[img->type]           += currMB->bitcounter[
        BITS_COEFF_UV_MB];
54
55     if (inp->symbol_mode == UVLC)
56         stat->bit_ctr += currMB->bitcounter[BITS_TOTAL_MB];
57     if (img->type==INTRA_IMG)
58         ++stat->mode_use_intra[img->mb_mode];
59     else
60         if (img->type != B_IMG)
61             ++stat->mode_use_inter[img->mb_mode];
62         else
63             ++stat->mode_use_Bframe[img->mb_mode];
64
65     /* Update coordinates of macroblock */
66     img->mb_x++;
67     if (img->mb_x == number_mb_per_row) /* next row of MBs */
68     {
69         img->mb_x = 0; /* start processing of next row */
70         img->mb_y++;
71     }
72     img->current_mb_nr++;
73
74
75     /* Define vertical positions */
76     img->block_y = img->mb_y * BLOCK_SIZE;      /* vertical luma
        block position */
77     img->pix_y    = img->mb_y * MB_BLOCK_SIZE;  /* vertical luma
        macroblock position */
78     img->pix_c_y = img->mb_y * MB_BLOCK_SIZE/2; /* vertical chroma
        macroblock position */
79
80     if (img->type != B_IMG)
81         if (inp->intra_upd > 0 && img->mb_y <= img->mb_y_intra)
82             img->height_err=(img->mb_y_intra*16)+15; /* for extra
        intra MB */
83     else
84         img->height_err=img->height-1;
85
86     /* Define horizontal positions */
87     img->block_x = img->mb_x * BLOCK_SIZE;      /* luma
        block */
88     img->pix_x    = img->mb_x * MB_BLOCK_SIZE;  /* luma
        pixel */
89     img->block_c_x = img->mb_x * BLOCK_SIZE/2;  /* chroma
        block */
90     img->pix_c_x    = img->mb_x * MB_BLOCK_SIZE/2; /* chroma
        pixel */
91
92     /* Statistics */
93     if (img->type == INTER_IMG)
94     {
95         ++stat->quant0;
96         stat->quant1 += img->qp; /* to find average quant for
        inter frames */
97     }
98 }
99
100 /******
101 **
102 *
```

```

102  *   Name :           start_macroblock()
103  *
104  *   Description: initializes the current macroblock
105  *
106  *****/
107 void start_macroblock(struct img_par *img, struct inp_par *inp, struct
stat_par *stat)
108 {
109     int i,j,k,l;
110     int x,y;
111     int use_bitstream_backing = (inp->slice_mode == FIXED_RATE || inp
->slice_mode == CALLBACK);
112     Macroblock *currMB = &img->mb_data[img->current_mb_nr];
113     Slice *curr_slice = img->currentSlice;
114     DataPartition *dataPart;
115     Bitstream *currStream;
116
117     if(use_bitstream_backing)
118     {
119         /* Save loopb and loopc in the case of recoding this
macroblock */
120         for(x = 0 ; x<6 ; x++)
121             for(y = 0 ; y<6 ; y++)
122                 img->tmp_loop_Y[x][y] = loopb[img->block_x+x][img->
block_y+y];
123         for(x=0 ; x<4 ; x++)
124             for(y=0 ; y<4 ; y++)
125                 img->tmp_loop_UV[x][y] = loopc[img->block_x/2+x][img->
block_y/2+y];
126
127         /* Keep the current state of the bitstreams */
128         for (i=0; i<curr_slice->max_part_nr; i++)
129         {
130             dataPart = &(curr_slice->partArr[i]);
131             currStream = dataPart->bitstream;
132             currStream->stored_bits_to_go = currStream->bits_to_go;
133             currStream->stored_byte_pos = currStream->byte_pos;
134             currStream->stored_byte_buf = currStream->byte_buf;
135         }
136     }
137
138     /* Save the slice number of this macroblock. When the macroblock
below
*/
139     /* is coded it will use this to decide if prediction for above is
possible */
140     img->slice_numbers[img->current_mb_nr] = img->current_slice_nr;
141
142     /* Save the slice and macroblock number of the current MB */
143     currMB->slice_nr = img->current_slice_nr;
144
145     /* Initialize counter for MB symbols */
146     currMB->currSEnr=0;
147
148     /* If MB is next to a slice boundary, mark neighboring blocks
unavailable for prediction */
149     CheckAvailabilityOfNeighbors(img);
150
151     /* Reset vectors before doing motion search in motion_search(). */
152     if (img->type != B_IMG)
153     {
154         for (k=0; k < 2; k++)
155         {
156             for (j=0; j < BLOCK_MULTIPLE; j++)
157                 for (i=0; i < BLOCK_MULTIPLE; i++)
158                     tmp_mv[k][img->block_y+j][img->block_x+i+4]=0;

```

```

159     }
160 }
161
162 /* Reset syntax element entries in MB struct */
163 currMB->mb_type = 0;
164 currMB->ref_frame = 0;
165 currMB->cbp = 0;
166
167 for (l=0; l < 2; l++)
168     for (j=0; j < BLOCK_MULTIPLE; j++)
169         for (i=0; i < BLOCK_MULTIPLE; i++)
170             for (k=0; k < 2; k++)
171                 currMB->mvd[l][j][i][k] = 0;
172
173 for (i=0; i < (BLOCK_MULTIPLE*BLOCK_MULTIPLE); i++)
174     currMB->intra_pred_modes[i] = 0;
175
176
177 /* Initialize bitcounters for this macroblock */
178 if (img->current_mb_nr == 0) /* No slice header to account for */
179 {
180     currMB->bitcounter[BITS_HEADER_MB] = 0;
181 }
182 else
183     if (img->slice_numbers[img->current_mb_nr] == img->
184         slice_numbers[img->current_mb_nr-1]) /* current MB belongs to
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2
```

```

212     Bitstream *currStream;
213
214     switch(inp->slice_mode)
215     {
216         case NO_SLICES:
217             *recode_macroblock = FALSE;
218             if ((mb_nr+1) == total_number_mb) /* maximum number of
219             MBs */
220                 *end_of_slice = TRUE;
221             break;
222         case FIXED_MB:
223             /* For slice mode one, check if a new slice boundary
224             follows */
225             *recode_macroblock = FALSE;
226             if ( ((mb_nr+1) % inp->slice_argument == 0) || ((mb_nr+1)
227             == total_number_mb) )
228             {
229                 *end_of_slice = TRUE;
230             }
231             break;
232
233             /* For slice modes two and three, check if coding of this
234             macroblock */
235             /* resulted in too many bits for this slice. If so, indicate
236             slice */
237             /* boundary before this macroblock and code the macroblock
238             again */
239             case FIXED_RATE:
240                 if (mb_nr > 0 && img->mb_data[mb_nr-1].slice_nr == img->
241                 current_slice_nr)
242                 {
243                     if (stat->bit_slice > inp->slice_argument*8)
244                     {
245                         //printf("MB: %d\tBits:
246                         %d\n",mb_nr,stat->bit_slice);
247                         *recode_macroblock = TRUE;
248                         *end_of_slice = TRUE;
249                     }
250                 }
251                 if ( (*recode_macroblock == FALSE) && ((mb_nr+1) ==
252                 total_number_mb) ) /* maximum number of MBs */
253                     *end_of_slice = TRUE;
254                 break;
255         case CALLBACK:
256             if (mb_nr > 0 && img->mb_data[mb_nr-1].slice_nr == img->
257             current_slice_nr)
258             {
259                 if (curr_slice->slice_too_big(stat->bit_slice))
260                 {
261                     *recode_macroblock = TRUE;
262                     *end_of_slice = TRUE;
263                 }
264             }
265             if ( (*recode_macroblock == FALSE) && ((mb_nr+1) ==
266             total_number_mb) ) /* maximum number of MBs */
267                 *end_of_slice = TRUE;
268             break;
269         default:
270             fprintf(stderr, "Slice Mode %d not supported", inp->
271             slice_mode);
272             exit(1);
273     }
274
275     if(*recode_macroblock == TRUE)
276     {
277         /* Restore the state of the bitstreams */

```

```

266         for (i=0; i<curr_slice->max_part_nr; i++)
267         {
268             dataPart = &(curr_slice->partArr[i]);
269             currStream = dataPart->bitstream;
270             currStream->bits_to_go = currStream->stored_bits_to_go;
271             currStream->byte_pos = currStream->
                stored_byte_pos;
272             currStream->byte_buf = currStream->stored_byte_buf;
273         }
274
275         /* Restore loopb and loopc before coding the MB again*/
276         for(x = 0 ; x<6 ; x++) for(y = 0 ; y<6 ; y++)
277             loopb[img->block_x+x][img->block_y+y] = img->tmp_loop_Y[x
                ][y];
278         for(x=0 ; x<4 ; x++) for(y=0 ; y<4 ; y++)
279             loopc[img->block_x/2+x][img->block_y/2+y] = img->
                tmp_loop_UV[x][y];
280     }
281
282 }
283
284 /*****
285  *
286  * Name :      CheckAvailabilityOfNeighbors(struct img_par *img)
287  *
288  * Description: Checks the availability of neighboring macroblocks of
289  *               the current macroblock for prediction
290  *               and context determination;
291  *               marks the unavailable MBs for intra
292  *               prediction in the
293  *               ipredmode-array by -1. Only
294  *               neighboring MBs in the causal
295  *               past of the current MB are checked.
296  *
297  *****/
298 void CheckAvailabilityOfNeighbors(struct img_par *img)
299 {
300     int i,j;
301     const int mb_width = img->width/MB_BLOCK_SIZE;
302     const int mb_nr = img->current_mb_nr;
303     Macroblock *currMB = &img->mb_data[mb_nr];
304
305     /* mark all neighbors as unavailable */
306     for (i=0; i<3; i++)
307         for (j=0; j<3; j++)
308             img->mb_data[mb_nr].mb_available[i][j]=NULL;
309     img->mb_data[mb_nr].mb_available[1][1]=currMB; /* current MB */
310
311     /* Check MB to the left */
312     if(img->pix_x >= MB_BLOCK_SIZE)
313     {
314         if(currMB->slice_nr != img->mb_data[mb_nr-1].slice_nr)
315         {
316             img->ipredmode[img->block_x][img->block_y+1] = -1;
317             img->ipredmode[img->block_x][img->block_y+2] = -1;
318             img->ipredmode[img->block_x][img->block_y+3] = -1;
319             img->ipredmode[img->block_x][img->block_y+4] = -1;
320         } else
321             currMB->mb_available[1][0]=&(img->mb_data[mb_nr-1]);
322     }
323
324     /* Check MB above */
325     if(img->pix_y >= MB_BLOCK_SIZE)

```

```

324     {
325         if(currMB->slice_nr != img->mb_data[mb_nr-mb_width].slice_nr)
326         {
327             img->ipredmode[img->block_x+1][img->block_y] = -1;
328             img->ipredmode[img->block_x+2][img->block_y] = -1;
329             img->ipredmode[img->block_x+3][img->block_y] = -1;
330             img->ipredmode[img->block_x+4][img->block_y] = -1;
331         } else
332             currMB->mb_available[0][1]=&(img->mb_data[mb_nr-
mb_width]);
333     }
334
335     /* Check MB left above */
336     if(img->pix_x >= MB_BLOCK_SIZE && img->pix_y >= MB_BLOCK_SIZE )
337     {
338         if(currMB->slice_nr == img->mb_data[mb_nr-mb_width-1].slice_nr)
339             img->mb_data[mb_nr].mb_available[0][0]=&(img->mb_data[
mb_nr-mb_width-1]);
340     }
341
342     /* Check MB right above */
343     if(img->pix_y >= MB_BLOCK_SIZE && img->pix_x < (img->width-
MB_BLOCK_SIZE ))
344     {
345         if(currMB->slice_nr == img->mb_data[mb_nr-mb_width+1].slice_nr)
346             currMB->mb_available[0][1]=&(img->mb_data[mb_nr-mb_width+1
]);
347     }
348 }
349
350 /*****
351  *
352  *   Name :           MakeIntraPrediction()
353  *
354  *   Description: Performs 4x4 and 16x16 intra prediction and transform
coding
355  *                               of the prediction residue. The
routine returns the best cost;
356  *                               current cbp (for LUMA only) and intra
pred modes are affected
357  *
358  *****/
359 int MakeIntraPrediction(struct img_par *img,struct inp_par *inp, int *
intra_pred_mode_2)
360 {
361
362     int i,j;
363     int block_x, block_y;
364     int best_ipmode;
365     int tot_intra_sad, tot_intra_sad2, best_intra_sad,
current_intra_sad;
366     int coeff_cost; // not used
367     int pic_pix_y,pic_pix_x,pic_block_y,pic_block_x;
368     int last_ipred; /* keeps last
chosen intra prediction mode for 4x4 intra pred */
369     int ipmode; /* intra pred mode */
370     int nonzero; /* keep track of nonzero
coefficients */
371     int cbp_mask;
372     Macroblock *currMB = &img->mb_data[img->current_mb_nr];
373
374     /* start making 4x4 intra prediction */
375     currMB->cbp=0;
376     img->mb_data[img->current_mb_nr].intraOrInter = INTRA_MB_4x4;

```

```

377
378     tot_intra_sad=QP2QUANT[img->qp]*24; /* sum of intra sad values,
start with a 'handicap' */
379
380     for(block_y = 0 ; block_y < MB_BLOCK_SIZE ; block_y +=
BLOCK_MULTIPLE)
381     {
382
383         pic_pix_y=img->pix_y+block_y;
384         pic_block_y=pic_pix_y/BLOCK_SIZE;
385
386         for(block_x = 0 ; block_x < MB_BLOCK_SIZE ; block_x +=
BLOCK_MULTIPLE)
387         {
388
389
390             cbp_mask=(1<<(2*(block_y/8)+block_x/8));
391
392
393             pic_pix_x=img->pix_x+block_x;
394             pic_block_x=pic_pix_x/BLOCK_SIZE;
395
396             /*
397             intrapred_luma() makes and returns 4x4 blocks with all 5
intra prediction modes.
398             Notice that some modes are not possible at frame edges.
399             */
400             intrapred_luma(img,pic_pix_x,pic_pix_y);
401
402
403             best_intra_sad=MAX_VALUE; /* initial value, will be
modified below */
404             img->imod = INTRA_MB_OLD; /* for now mode set to intra,
may be changed in motion_search() */
405             /* DM: has to be removed */
406
407             for (ipmode=0; ipmode < NO_INTRA_PMODE; ipmode++) /*
all intra prediction modes */
408             {
409                 /* Horizontal pred from Y neighbour pix , vertical
use X pix, diagonal needs both */
410                 if (ipmode==DC_PRED||ipmode==HOR_PRED||img->ipredmode[
pic_block_x+1][pic_block_y] >= 0)/* DC or vert pred
or hor edge*/
411                 {
412                     if (ipmode==DC_PRED||ipmode==VERT_PRED||img->
ipredmode[pic_block_x][pic_block_y+1] >= 0)/* DC
or hor pred or vert edge*/
413                     {
414                         for (j=0; j < BLOCK_SIZE; j++)
415                         {
416                             for (i=0; i < BLOCK_SIZE; i++)
417                                 img->m7[i][j]=imgY_org[pic_pix_y+j][
pic_pix_x+i]-img->mprp[ipmode][j][i];
/* find diff */
418                         }
419                         current_intra_sad=QP2QUANT[img->qp]*PRED_IPRED
[img->ipredmode[pic_block_x+1][pic_block_y]+1
][img->ipredmode[pic_block_x][pic_block_y+1]+1
][ipmode]*2;
420
421                         current_intra_sad += find_sad(img,inp->
hadamard); /* add the start 'handicap' and
the computed SAD */
422
423                         if (current_intra_sad < best_intra_sad)

```

```

424         {
425             best_intra_sad=current_intra_sad;
426             best_ipmode=ipmode;
427
428             for (j=0; j < BLOCK_SIZE; j++)
429                 for (i=0; i < BLOCK_SIZE; i++)
430                     img->mpr[i+block_x][j+block_y]=img
                        ->mpr[ipmode][j][i]; /*
                        store the currently best intra
                        prediction block*/
431         }
432     }
433 }
434
435 tot_intra_sad += best_intra_sad;
436
437 img->ipredmode[pic_block_x+1][pic_block_y+1]=best_ipmode;
438
439 if (pic_block_x & 1 == 1) /* just even blocks, two and
two predmodes are sent together */
440 {
441     currMB->intra_pred_modes[block_x/4+block_y]=PRED_IPRED
[img->ipredmode[pic_block_x+1][pic_block_y+1][img->
ipredmode[pic_block_x][pic_block_y+1]+1][best_ipmode];
442     currMB->intra_pred_modes[block_x/4-1+block_y]=
last_ipred;
443 }
444 last_ipred=PRED_IPRED[img->ipredmode[pic_block_x+1][
pic_block_y+1][img->ipredmode[pic_block_x][pic_block_y+1
]+1][best_ipmode];
445
446 /* Make difference from prediction to be transformed*/
447 for (j=0; j < BLOCK_SIZE; j++)
448     for (i=0; i < BLOCK_SIZE; i++)
449     {
450         img->m7[i][j] =imgY_org[img->pix_y+block_y+j][img
->pix_x+block_x+i] - img->mpr[i+block_x][j+block_y
];
451     }
452
453 nonzero = dct_luma(block_x,block_y,&coeff_cost,img);
454
455 if (nonzero)
456 {
457     currMB->cbp |= cbp_mask; /* set coded block
pattern if there are nonzero coeffs */
458 }
459 }
460 }
461
462 /* 16x16 intra prediction */
463 intrapred_luma_2(img); /* make
intra pred for the new 4 modes */
464 tot_intra_sad2 = find_sad2(img,intra_pred_mode_2); /* find best
SAD for new modes */
465
466 if (tot_intra_sad2<tot_intra_sad)
467 {
468     currMB->cbp = 0;
469
470     /* cbp for
16x16 LUMA is signaled by the MB-mode */
471     tot_intra_sad = tot_intra_sad2; /* update best intra sad if
necessary */
472     img->imod = INTRA_MB_NEW; /* one of the new modes is
used */
473     img->mb_data[img->current_mb_nr].intraOrInter = INTRA_MB_16x16;

```

```

472         dct_luma2(img,*intra_pred_mode_2);
473         for (i=0;i<4;i++)
474             for (j=0;j<4;j++)
475                 img->ipredmode[img->block_x+i+1][img->block_y+j+1]=0;
476     }
477     return tot_intra_sad;
478 }
479
480 /*****
481  *
482  *   Name :           LumaResidualCoding_P()
483  *
484  *   Description: Performs DCT, R-D constrained quantization, run/level
485  *                   pre-coding and IDCT for the
486  *                   MC-compensated MB residue
487  *                   of P-frame; current cbp (for LUMA
488  *                   only) is affected
489  *
490  *****/
491 void LumaResidualCoding_P(struct img_par *img,struct inp_par *inp)
492 {
493     int i,j;
494     int block_x, block_y;
495     int pic_pix_y,pic_pix_x,pic_block_y,pic_block_x;
496     int ii4,i2,jj4,j2;
497     int sum_cnt_nonz;
498     int mb_x, mb_y;
499     int cbp_mask;
500     int coeff_cost;
501     int nonzero;
502     Macroblock *currMB = &img->mb_data[img->current_mb_nr];
503
504     currMB->cbp=0;
505     sum_cnt_nonz=0;
506     for (mb_y=0; mb_y < MB_BLOCK_SIZE; mb_y += BLOCK_SIZE*2)
507     {
508         for (mb_x=0; mb_x < MB_BLOCK_SIZE; mb_x += BLOCK_SIZE*2)
509         {
510             cbp_mask=(1<<(mb_x/8+mb_y/4));
511             coeff_cost=0;
512             for (block_y=mb_y; block_y < mb_y+BLOCK_SIZE*2; block_y +=
513                 BLOCK_SIZE)
514             {
515                 pic_pix_y=img->pix_y+block_y;
516                 pic_block_y=pic_pix_y/BLOCK_SIZE;
517
518                 for (block_x=mb_x; block_x < mb_x+BLOCK_SIZE*2;
519                     block_x += BLOCK_SIZE)
520                 {
521                     pic_pix_x=img->pix_x+block_x;
522                     pic_block_x=pic_pix_x/BLOCK_SIZE;
523
524                     img->ipredmode[pic_block_x+1][pic_block_y+1]=0;
525                     ii4=(img->pix_x+block_x)*4+tmp_mv[0][pic_block_y][
526                         pic_block_x+4];
527                     jj4=(img->pix_y+block_y)*4+tmp_mv[1][pic_block_y][
528                         pic_block_x+4];
529                     for (j=0;j<4;j++)
530                     {
531                         j2=j*4;
532                         for (i=0;i<4;i++)
533                         {

```

```

530             i2=i*4;
531             img->mpr[i+block_x][j+block_y]=mref[img->
multiframe_no][jj4+j2][ii4+i2];
532         }
533     }
534
535     for (j=0; j < BLOCK_SIZE; j++)
536     {
537         for (i=0; i < BLOCK_SIZE; i++)
538         {
539             img->m7[i][j] =imgY_org[img->pix_y+block_y
+j][img->pix_x+block_x+i] - img->mpr[i+
block_x][j+block_y];
540         }
541     }
542     nonzero=dct_luma(block_x,block_y,&coeff_cost,img);
543     if (nonzero)
544     {
545         currMB->cbp |= cbp_mask;
546     }
547 }
548 }
549
550 /*
551 The purpose of the action below is to prevent that single
or 'expensive' coefficients are coded.
552 With 4x4 transform there is larger chance that a single
coefficient in a 8x8 or 16x16 block may be nonzero.
553 A single small (level=1) coefficient in a 8x8 block will
cost: 3 or more bits for the coefficient,
554 4 bits for EOBs for the 4x4 blocks,possibly also more
bits for CBP. Hence the total 'cost' of that single
555 coefficient will typically be 10-12 bits which in a RD
consideration is too much to justify the distortion
improvement.
556 The action below is to watch such 'single' coefficients
and set the reconstructed block equal to the prediction
according
557 to a given criterium. The action is taken only for inter
luma blocks.
558
559 Notice that this is a pure encoder issue and hence does
not have any implication on the standard.
560 coeff_cost is a parameter set in dct_luma() and
accumulated for each 8x8 block. If level=1 for a
coefficient,
561 coeff_cost is increased by a number depending on RUN for
that coefficient.The numbers are (see also dct_luma()):
3,2,2,1,1,1,0,0,...
562 when RUN equals 0,1,2,3,4,5,6, etc.
563 If level >1 coeff_cost is increased by 9 (or any number
above 3). The threshold is set to 3. This means for
example:
564 1: If there is one coefficient with (RUN,level)=(0,1) in
a 8x8 block this coefficient is discarded.
565 2: If there are two coefficients with (RUN,level)=(1,1)
and (4,1) the coefficients are also discarded
566 sum_cnt_nonz is the accumulation of coeff_cost over a
whole macro block. If sum_cnt_nonz is 5 or less for the
whole MB,
567 all nonzero coefficients are discarded for the MB and the
reconstructed block is set equal to the prediction.
568 */
569
570 if (coeff_cost > 3)
571 {

```

```

572         sum_cnt_nonz += coeff_cost;
573     }
574     else /*discard */
575     {
576         currMB->cbp &= (63-cbp_mask);
577         for (i=mb_x; i < mb_x+BLOCK_SIZE*2; i++)
578         {
579             for (j=mb_y; j < mb_y+BLOCK_SIZE*2; j++)
580             {
581                 imgY[img->pix_y+j][img->pix_x+i]=img->mpr[i][j];
582             }
583         }
584     }
585 }
586
587
588 if (sum_cnt_nonz <= 5 )
589 {
590     currMB->cbp &= 48; /* mask bit 4 and 5 */
591     for (i=0; i < MB_BLOCK_SIZE; i++)
592     {
593         for (j=0; j < MB_BLOCK_SIZE; j++)
594         {
595             imgY[img->pix_y+j][img->pix_x+i]=img->mpr[i][j];
596         }
597     }
598 }
599 }
600
601 /*****
602  *
603  *   Name :           ChromaCoding_P()
604  *
605  *   Description: Performs DCT, quantization, run/level pre-coding and
606  *                   IDCT
607  *                   for the chrominance of a I- of P-frame macroblock;
608  *                   current cbp and cr_cbp are affected
609  *
610  *****/
611 void ChromaCoding_P(int *cr_cbp, struct img_par *img)
612 {
613     int i, j;
614     int uv, ii,jj,ii0,jj0,ii1,jj1,if1,jf1,if0,jf0;
615     int pic_block_y, pic_block_x;
616     Macroblock *currMB = &img->mb_data[img->current_mb_nr];
617
618     *cr_cbp=0;
619     for (uv=0; uv < 2; uv++)
620     {
621         if (img->imod == INTRA_MB_OLD || img->imod == INTRA_MB_NEW)
622         {
623             intrapred_chroma(img,img->pix_c_x,img->pix_c_y,uv);
624         }
625         else
626         {
627             for (j=0; j < MB_BLOCK_SIZE/2; j++)
628             {
629                 pic_block_y=(img->pix_c_y+j)/2;
630                 for (i=0; i < MB_BLOCK_SIZE/2; i++)
631                 {
632                     pic_block_x=(img->pix_c_x+i)/2;
633                     ii=(img->pix_c_x+i)*8+tmp_mv[0][pic_block_y][
634                         pic_block_x+4];

```

```

633         jj=(img->pix_c_y+j)*8+tmp_mv[1][pic_block_y][
        pic_block_x+4];
634         ii0=ii/8;
635         jj0=jj/8;
636         ii1=(ii+7)/8;
637         jj1=(jj+7)/8;
638         if1=(ii & 7);
639         jf1=(jj & 7);
640         if0=8-if1;
641         jf0=8-jf1;
642         img->mpr[i][j]=(if0*jf0*mcef[img->multiframe_no][uv
        ][jj0][ii0]+
643         if1*jf0*mcef[img->multiframe_no][uv][jj0][ii1]+
644         if0*jf1*mcef[img->multiframe_no][uv][jj1][ii0]+
645         if1*jf1*mcef[img->multiframe_no][uv][jj1][ii1]+
        32)/64;
646         img->m7[i][j]=imgUV_org[uv][img->pix_c_y+j][img->
        pix_c_x+i]-img->mpr[i][j];
647     }
648 }
649 }
650     *cr_cbp=dct_chroma(uv,*cr_cbp,img);
651 }
652     currMB->cbp += *cr_cbp*16;
653 }
654
655 /*****
656 *
657 *   Name :           SetRefFrameInfo_P()
658 *
659 *   Description: Set reference frame information in global arrays
660 *                   depending on mode decision. Used for
        motion vector prediction.
661 *
662 *****/
663 void SetRefFrameInfo_P(struct img_par *img)
664 {
665     int i,j;
666
667     if (img->imod == INTRA_MB_OLD || img->imod == INTRA_MB_NEW)
668     {
669         /* Set the reference frame information for motion vector
        prediction as unavailable */
670         for (j = 0;j < BLOCK_MULTIPLE;j++)
671         {
672             for (i = 0;i < BLOCK_MULTIPLE;i++)
673             {
674                 refFrArr[img->block_y+j][img->block_x+i] = -1;
675             }
676         }
677     }
678     else
679     {
680         /* Set the reference frame information for motion vector
        prediction */
681         for (j = 0;j < BLOCK_MULTIPLE;j++)
682         {
683             for (i = 0;i < BLOCK_MULTIPLE;i++)
684             {
685                 refFrArr[img->block_y+j][img->block_x+i] = img->
        mb_data[img->current_mb_nr].ref_frame;
686             }
687         }
688     }

```

```

689     }
690
691
692     /*****
693     *
694     *   Name :           SetLoopfilterStrength_P()
695     *
696     *   Description: Set the filter strength for a macroblock of a I- or
697     *   P-frame
698     *
699     *****/
700 void SetLoopfilterStrength_P(struct img_par *img)
701 {
702     int i,j;
703     int ii,jj;
704     int i3,j3,mvDiffX,mvDiffY;
705
706     if (img->imod == INTRA_MB_OLD || img->imod == INTRA_MB_NEW)
707     {
708         for (i=0;i<BLOCK_MULTIPLE;i++)
709         {
710             ii=img->block_x+i;
711             i3=ii/2;
712             for (j=0;j<BLOCK_MULTIPLE;j++)
713             {
714                 jj=img->block_y+j;
715                 j3=jj/2;
716                 loopb[ii+1][jj+1]=3;
717                 loopb[ii ][jj+1]=max(loopb[ii ][jj+1],2);
718                 loopb[ii+1][jj ]=max(loopb[ii+1][jj ],2);
719                 loopb[ii+2][jj+1]=max(loopb[ii+2][jj+1],2);
720                 loopb[ii+1][jj+2]=max(loopb[ii+1][jj+2],2);
721
722                 loopc[i3+1][j3+1]=2;
723                 loopc[i3 ][j3+1]=max(loopc[i3 ][j3+1],1);
724                 loopc[i3+1][j3 ]=max(loopc[i3+1][j3 ],1);
725                 loopc[i3+2][j3+1]=max(loopc[i3+2][j3+1],1);
726                 loopc[i3+1][j3+2]=max(loopc[i3+1][j3+2],1);
727             }
728         }
729     else
730     {
731         for (i=0;i<4;i++)
732         {
733             ii=img->block_x+i;
734             i3=ii/2;
735             for (j=0;j<4;j++)
736             {
737                 jj=img->block_y+j;
738                 j3=jj/2;
739
740                 mvDiffX = tmp_mv[0][jj][ii+4] - tmp_mv[0][jj][ii-1+4];
741                 mvDiffY = tmp_mv[1][jj][ii+4] - tmp_mv[1][jj][ii-1+4];
742
743                 if((mvDiffX*mvDiffX >= 16 || mvDiffY*mvDiffY >= 16) &&
744                     ii > 0)
745                 {
746                     loopb[ii ][jj+1]=max(loopb[ii ][jj+1],1);
747                     loopb[ii+1][jj+1]=max(loopb[ii+1][jj+1],1);
748                     loopc[i3 ][j3+1]=max(loopc[i3 ][j3+1],1);
749                     loopc[i3+1][j3+1]=max(loopc[i3+1][j3+1],1);
750

```

```

751         if(jj > 0) /*GH: bug fix to avoid tmp_mv[][-1][ii+4]*/
752         {
753             mvDiffX = tmp_mv[0][jj][ii+4] - tmp_mv[0][jj-1][ii+4];
754             mvDiffY = tmp_mv[1][jj][ii+4] - tmp_mv[1][jj-1][ii+4];
755
756             if(mvDiffX*mvDiffX >= 16 || mvDiffY*mvDiffY >= 16)
757             {
758                 loopb[ii+1][jj] = max(loopb[ii+1][jj], 1);
759                 loopb[ii+1][jj+1] = max(loopb[ii+1][jj+1], 1);
760                 loopc[i3+1][j3] = max(loopc[i3+1][j3], 1);
761                 loopc[i3+1][j3+1] = max(loopc[i3+1][j3+1], 1);
762             }
763         }
764     }
765 }
766 }
767 }
768
769 /**
770 *
771 * Name : encode_one_macroblock()
772 *
773 * Description: Encode one macroblock depending on chosen picture type
774 *
775 ****
776 void encode_one_macroblock(struct img_par *img, struct inp_par *inp,
777 struct stat_par *stat)
778 {
779     int cr_cbp; /* chroma coded block pattern */
780     int tot_intra_sad;
781     int intra_pred_mode_2; /* best 16x16 intra mode */
782
783     Macroblock *currMB = &img->mb_data[img->current_mb_nr];
784
785     /* Intra Prediction */
786     tot_intra_sad = MakeIntraPrediction(img, inp, &intra_pred_mode_2);
787
788     if (img->type != B_IMG) /* I- or P-frame */
789     {
790         if ((img->mb_y == img->mb_y_upd && img->mb_y_upd != img->
791             mb_y_intra) || img->type == INTRA_IMG)
792         {
793             img->mb_mode = 8*img->type + img->imod; /* Intra mode: set
794                 if intra image or if intra GOB for error robustness */
795         }
796         else
797         {
798             currMB->ref_frame = motion_search(inp, img, tot_intra_sad);
799             /* Motion vector search for P-frames */
800         }
801     }
802     else /*
803         B-frame */
804     {
805         currMB->ref_frame = motion_search_Bframe(tot_intra_sad, img,
806             inp); /* Motion vector search for B-frames */
807
808         if (img->type == B_IMG) /* B-frame */
809         {
810             /* Residual coding of Luminance (only for B-modes) */
811             LumaResidualCoding_B(img);
812         }
813     }
814 }

```

```

806         /* Coding of Chrominance */
807         ChromaCoding_B(&cr_cbp, img);
808
809         /* Set loop filter strength depending on mode decision */
810         SetLoopfilterStrength_B(img);
811
812         /* Set reference frame information for motion vector
      prediction of future MBs */
813         SetRefFrameInfo_B(img);
814     }
815     else /* I-
      or P-frame */
816     {
817         if (currMB->intraOrInter == INTER_MB)
818             LumaResidualCoding_P(img, inp); /* Residual
      coding of Luminance (only for inter modes) */
819
820         /* Coding of Luma in intra mode is done implicitly in
      MakeIntraPrediction */
821         /* Coding of Chrominance */
822         ChromaCoding_P(&cr_cbp, img);
823
824         /* Set loop filter strength depending on mode decision */
825         SetLoopfilterStrength_P(img);
826
827         /* Set reference frame information for motion vector
      prediction of future MBs */
828         SetRefFrameInfo_P(img);
829
830         /* Check if a MB is skipped (no coeffs. only 0-vectors and
      prediction from the most recent frame) */
831         if (img->mb_mode==M16x16_MB && currMB->intraOrInter ==
      INTER_MB && (currMB->cbp == 0)
832             && tmp_mv[0][img->block_y][img->block_x+4]==0 &&
      tmp_mv[1][img->block_y][img->block_x+4]==0
833             && (currMB->ref_frame == 0))
834             img->mb_mode=COPY_MB;
835     }
836
837     /* Set 16x16 intra mode and make "intra CBP" */
838     if (img->imod==INTRA_MB_NEW)
839         img->mb_mode += intra_pred_mode_2 + 4*cr_cbp + 12*img->kac;
840
841 }
842
843 /*****
      **
844 *
845 *   Name :       write_one_macroblock()
846 *
847 *   Description: Passes the chosen syntax elements to the NAL
848 *
849 *****/
850 void write_one_macroblock(struct img_par *img, struct inp_par *inp,
      struct stat_par *stat)
851 {
852     int i;
853     int mb_nr = img->current_mb_nr;
854     SyntaxElement *currSE = img->MB_SyntaxElements;
855     Macroblock *currMB = &img->mb_data[mb_nr];
856     int *bitCount = currMB->bitcounter;
857     Slice *currSlice = img->currentSlice;
858     DataPartition *dataPart;
859     int *partMap = assignSE2partition[inp->partition_mode];

```

```

860
861
862     /* Store imod for further use */
863     currMB->mb_imode = img->imod;
864
865     /* Store mb_mode for further use */
866     currMB->mb_type = (currSE->value1 = img->mb_mode);
867
868     /* Bits for mode */
869     if (inp->symbol_mode == UVLC)
870         currSE->mapping = n_linfo2;
871     else
872         currSE->writing = writeMB_typeInfo2Buffer_CABAC;
873     currSE->type = SE_MBTYPE;
874
875
876     /* choose the appropriate data partition */
877     if (img->type != B_IMG)
878     {
879 #if TRACE
880         sprintf(currSE->tracestring, "MB mode(%2d,%2d) = %3d",img->
            mb_x, img->mb_y,img->mb_mode);
881 #endif
882         dataPart = &(currSlice->partArr[partMap[SE_MBTYPE]]);
883     }
884     else
885     {
886 #if TRACE
887         sprintf(currSE->tracestring, "B_MB mode(%2d,%2d) = %3d",img->
            mb_x, img->mb_y, img->mb_mode);
888 #endif
889         dataPart = &(currSlice->partArr[partMap[SE_BFRAME]]);
890     }
891     dataPart->writeSyntaxElement( currSE, img, inp, dataPart);
892     bitCount[BITS_HEADER_MB]+=currSE->len;
893
894     /* proceed to next SE */
895     currSE++;
896     currMB->currSEnr++;
897
898
899     /* Do nothing more if copy and inter mode */
900     if (img->mb_mode != COPY_MB || currMB->intraOrInter != INTER_MB ||
        img->type == B_IMG)
901     {
902
903         /* Bits for intra prediction modes*/
904         if (img->imod == INTRA_MB_OLD)
905         {
906             for (i=0; i < MB_BLOCK_SIZE/2; i++)
907             {
908                 currSE->value1 = currMB->intra_pred_modes[2*i];
909                 currSE->value2 = currMB->intra_pred_modes[2*i+1];
910                 if (inp->symbol_mode == UVLC)
911                     currSE->mapping = intrapred_linfo;
912                 else
913                     currSE->writing = writeIntraPredMode2Buffer_CABAC;
914                 currSE->type = SE_INTRAPREDMODE;
915
916                 /* choose the appropriate data partition */
917                 if (img->type != B_IMG)
918                 {
919 #if TRACE
920                     sprintf(currSE->tracestring, "Intra mode      =
                        %3d",IPRED_ORDER[currSE->value1][currSE->value2]);
921 #endif

```

```

922         dataPart = &(currSlice->partArr[partMap[
SE_INTRAPREDMODE]]);
923
924     }
925     else
926     {
927         #if TRACE
928             sprintf(currSE->tracestring, "B_Intra mode =
%3d\t", IPRED_ORDER[currSE->value1][currSE->value2
]);
929         #endif
930         dataPart = &(currSlice->partArr[partMap[SE_BFRAME
]);
931
932     }
933     dataPart->writeSyntaxElement( currSE, img, inp,
dataPart);
934     bitCount[BITS_COEFF_Y_MB]+=currSE->len;
935
936     /* proceed to next SE */
937     currSE++;
938     currMB->currSEnr++;
939 }
940 }
941 /* Bits for vector data*/
942 if (img->type != B_IMG)
943 {
944     if (currMB->intraOrInter == INTER_MB) /* inter */
945         writeMotionInfo2NAL_Pframe(img,inp);
946 }
947 else
948 {
949     if(img->imod != B_Direct)
950         writeMotionInfo2NAL_Bframe(img, inp);
951 }
952
953 /* Bits for CBP and Coefficients */
954 writeCBPandCoeffs2NAL(img,inp);
955 }
956 bitCount[BITS_TOTAL_MB] = bitCount[BITS_HEADER_MB] + bitCount[
BITS_COEFF_Y_MB] + bitCount[BITS_INTER_MB]
957
958                                     + bitCount[
BITS_CBP_MB] +
959                                     bitCount[
BITS_COEFF_UV_MB];
960
961 stat->bit_slice += bitCount[BITS_TOTAL_MB];
962 }
963
964 /*****
965 **
966 * Name : writeMotionInfo2NAL_Pframe()
967 *
968 * Description: Passes for a given MB of a P picture the reference
969 *
970 * parameter and the motion vectors to
971 * the NAL
972 *
973 *****/
974 void writeMotionInfo2NAL_Pframe(struct img_par *img,struct inp_par *
inp)
975 {
976     int i,j,k,l,m;
977     int step_h,step_v;

```

```

974     int curr_mvd;
975     int mb_nr = img->current_mb_nr;
976     Macroblock *currMB = &img->mb_data[mb_nr];
977     SyntaxElement *currSE = &img->MB_SyntaxElements[currMB->currSEnr];
978     int *bitCount = currMB->bitcounter;
979     Slice *currSlice = img->currentSlice;
980     DataPartition *dataPart;
981     int *partMap = assignSE2partition[inp->partition_mode];
982
983     /* If multiple ref. frames, write reference frame for the MB */
984     if (inp->no_multpred > 1)
985     {
986
987         currSE->value1 = currMB->ref_frame ;
988         currSE->type = SE_REFFRAME;
989         if (inp->symbol_mode == UVLC)
990             currSE->mapping = n_linfo2;
991         else
992             currSE->writing = writeRefFrame2Buffer_CABAC;
993         dataPart = &(currSlice->partArr[partMap[currSE->type]]);
994         dataPart->writeSyntaxElement( currSE, img, inp, dataPart);
995         bitCount[BITS_INTER_MB]+=currSE->len;
996         #if TRACE
997             sprintf(currSE->tracestring, "Reference frame no %d", currMB->
998                 ref_frame);
999         #endif
1000
1001         /* proceed to next SE */
1002         currSE++;
1003         currMB->currSEnr++;
1004     }
1005
1006     /* Write motion vectors */
1007     step_h=img->blc_size_h/BLOCK_SIZE;          /* horizontal
1008     stepsize */
1009     step_v=img->blc_size_v/BLOCK_SIZE;          /* vertical stepsize */
1010
1011     for (j=0; j < BLOCK_SIZE; j += step_v)
1012     {
1013         for (i=0; i < BLOCK_SIZE; i += step_h)
1014         {
1015             for (k=0; k < 2; k++)
1016             {
1017                 curr_mvd = tmp_mv[k][img->block_y+j][img->block_x+i+4
1018                     ]-img->mv[i][j][currMB->ref_frame][img->mb_mode][k];
1019
1020                 img->subblock_x = i; // position used for context
1021                 determination
1022                 img->subblock_y = j; // position used for context
1023                 determination
1024                 currSE->value1 = curr_mvd;
1025                 /* store (oversampled) mvd */
1026                 for (l=0; l < step_v; l++)
1027                     for (m=0; m < step_h; m++)
1028                         currMB->mvd[0][j+l][i+m][k] = curr_mvd;
1029                 currSE->value2 = k; // identifies the component; only
1030                 used for context determination
1031                 currSE->type = SE_MVD;
1032                 if (inp->symbol_mode == UVLC)
1033                     currSE->mapping = mvd_linfo2;
1034                 else
1035                     currSE->writing = writeMVD2Buffer_CABAC;
1036                 dataPart = &(currSlice->partArr[partMap[currSE->type
1037                     ]]);
1038                 dataPart->writeSyntaxElement( currSE, img, inp,

```

```

1032         dataPart);
1033         bitCount[BITS_INTER_MB]+=currSE->len;
1034         #if TRACE
1035             sprintf(currSE->tracestring, " MVD(%d) = %3d",k,
1036             curr_mvd);
1037         #endif
1038         /* proceed to next SE */
1039         currSE++;
1040         currMB->currSEnr++;
1041     }
1042 }
1043 }
1044 }
1045
1046 /*****
1047 *
1048 * Name :      writeCBPandCoeffs2NAL()
1049 *
1050 * Description: Passes coded block pattern and coefficients (run/level)
1051 *               to the NAL
1052 *
1053 *****/
1054 void writeCBPandCoeffs2NAL(struct img_par *img,struct inp_par *inp)
1055 {
1056     int i,j,k;
1057     int mb_y,mb_x;
1058     int level, run;
1059     int uv;
1060     int kk,kbeg,kend;
1061     int mb_nr = img->current_mb_nr;
1062     int ii,jj;
1063     int i1,j1, m2,jg2;
1064     Macroblock *currMB = &img->mb_data[mb_nr];
1065     const int cbp = currMB->cbp;
1066     SyntaxElement *currSE = &img->MB_SyntaxElements[currMB->currSEnr];
1067     int *bitCount = currMB->bitcounter;
1068     Slice *currSlice = img->currentSlice;
1069     DataPartition *dataPart;
1070     int *partMap = assignSE2partition[inp->partition_mode];
1071
1072     /* Bits for CBP */
1073     if (img->imod != INTRA_MB_NEW)
1074     {
1075         currSE->value1 = cbp;
1076         #if TRACE
1077             sprintf(currSE->tracestring, "CBP (%2d,%2d) = %3d",img->mb_x,
1078             img->mb_y, cbp);
1079         #endif
1080
1081         if (img->imod == INTRA_MB_OLD)
1082         {
1083             if (inp->symbol_mode == UVLC)
1084                 currSE->mapping = cbp_lininfo_intra;
1085             currSE->type = SE_CBP_INTRA;
1086         }
1087         else
1088         {
1089             if (inp->symbol_mode == UVLC)
1090                 currSE->mapping = cbp_lininfo_inter;
1091             currSE->type = SE_CBP_INTER;
1092

```

```

1093     }
1094
1095     if (inp->symbol_mode == CABAC)
1096         currSE->writing = writeCBP2Buffer_CABAC;
1097
1098     /* choose the appropriate data partition */
1099     if (img->type != B_IMG)
1100         dataPart = &(currSlice->partArr[partMap[currSE->type]]);
1101     else
1102         dataPart = &(currSlice->partArr[partMap[SE_BFRAME]]);
1103
1104     dataPart->writeSyntaxElement( currSE, img, inp, dataPart);
1105     bitCount[BITS_CBP_MB] += currSE->len;
1106
1107     /* proceed to next SE */
1108     currSE++;
1109     currMB->currSEnr++;
1110
1111     /* Bits for luma coefficients */
1112     for (mb_y=0; mb_y < 4; mb_y += 2)
1113     {
1114         for (mb_x=0; mb_x < 4; mb_x += 2)
1115         {
1116             for (j=mb_y; j < mb_y+2; j++)
1117             {
1118                 jj=j/2;
1119                 for (i=mb_x; i < mb_x+2; i++)
1120                 {
1121                     ii=i/2;
1122                     if ((cbp & (1<<(ii+jj*2))) != 0)
1123                         /* check for any
1124                         coefficients */
1125                     {
1126                         if (img->imod == INTRA_MB_OLD && img->qp <
1127                             24) /* double scan */
1128                         {
1129                             for(kk=0;kk<2;kk++)
1130                             {
1131                                 kbeg=kk*9;
1132                                 kend=kbeg+8;
1133                                 level=1; /* get inside loop */
1134
1135                                 for(k=kbeg;k <= kend && level !=0;
1136                                     k++)
1137                                 {
1138                                     level = currSE->value1 = img->
1139                                         cof[i][j][k][0][DOUBLE_SCAN];
1140                                     // level
1141                                     run = currSE->value2 = img->
1142                                         cof[i][j][k][1][DOUBLE_SCAN];
1143                                     // run
1144
1145                                     if (inp->symbol_mode == UVLC)
1146                                         currSE->mapping =
1147                                             levrn_linfo_intra;
1148                                     else
1149                                     {
1150                                         currSE->context = 0; //
1151                                         for choosing context model
1152                                         currSE->writing =
1153                                             writeRunLevel2Buffer_CABAC;
1154                                     }
1155
1156                                     if (k == kbeg)
1157                                     {

```

```

1148 currSE->type =
SE_LUM_DC_INTRA; /*
element is of type DC */

1149
1150 /* choose the appropriate
data partition */
1151 if (img->type != B_IMG)
1152     dataPart = &(currSlice
->partArr[partMap[
SE_LUM_DC_INTRA]]);
1153 else
1154     dataPart = &(currSlice
->partArr[partMap[
SE_BFRAME]]);
1155 }
1156 else
1157 {
1158     currSE->type =
SE_LUM_AC_INTRA; /*
element is of type AC */

1159
1160 /* choose the appropriate
data partition */
1161 if (img->type != B_IMG)
1162     dataPart = &(currSlice
->partArr[partMap[
SE_LUM_AC_INTRA]]);
1163 else
1164     dataPart = &(currSlice
->partArr[partMap[
SE_BFRAME]]);
1165
1166 }
1167 dataPart->writeSyntaxElement
( currSE, img, inp, dataPart
);
1168 bitCount[BITS_COEFF_Y_MB]+=
currSE->len;

1169 #if TRACE
1170 sprintf(currSE->tracestring,
"Luma dbl(%2d,%2d) level=%3d
Run=%2d",kk,k,level,run);

1171 #endif
1172
1173 /* proceed to next SE */
1174 currSE++;
1175 currMB->currSEnr++;
1176
1177 if (level!=0)
1178 {
1179     loopb[img->block_x+i+1][
img->block_y+j+1]=max(
loopb[img->block_x+i+1][
img->block_y+j+1],2);
1180
1181     loopb[img->block_x+i][
img->block_y+j+1]=max(
loopb[img->block_x+i][
img->block_y+j+1],1);
1182     loopb[img->block_x+i+1][
img->block_y+j]=max(
loopb[img->block_x+i+1][
img->block_y+j],1);
1183     loopb[img->block_x+i+2][
img->block_y+j+1]=max(
loopb[img->block_x+i+2][
img->block_y+j+1],1);

```

```

1183                                     loopb[img->block_x+i+1][
                                     img->block_y+j+2]=max(
                                     loopb[img->block_x+i+1][
                                     img->block_y+j+2],1);
1184                                     }
1185                                 }
1186                             }
1187                         }
1188                     else

                                     /* single scan */
1189                 {
1190
1191                     level=1; /* get inside loop */
1192                     for(k=0;k<=16 && level !=0; k++)
1193                     {
1194                         level = currSE->value1 = img->cof[
1195                             i][j][k][0][SINGLE_SCAN]; // level
1196                         run = currSE->value2 = img->cof[i
1197                             ][j][k][1][SINGLE_SCAN]; // run
1198
1199                         if (inp->symbol_mode == UVLC)
1200                             currSE->mapping =
1201                                 levrn_linfo_inter;
1202                         else
1203                             currSE->writing =
1204                                 writeRunLevel2Buffer_CABAC;
1205
1206                         if (k == 0)
1207                         {
1208                             if (img->imod == INTRA_MB_OLD
1209                                 || img->imod == INTRA_MB_NEW)
1210                             {
1211                                 currSE->context = 2; //
1212                                 for choosing context model
1213                                 currSE->type =
1214                                     SE_LUM_DC_INTRA;
1215                             }
1216                             else
1217                             {
1218                                 currSE->context = 1; //
1219                                 for choosing context model
1220                                 currSE->type =
1221                                     SE_LUM_DC_INTER;
1222                             }
1223                         }
1224                         else
1225                         {
1226                             if (img->imod == INTRA_MB_OLD
1227                                 || img->imod == INTRA_MB_NEW)
1228                             {
1229                                 currSE->context = 2; //
1230                                 for choosing context model
1231                                 currSE->type =
1232                                     SE_LUM_AC_INTRA;
1233                             }
1234                             else
1235                             {
1236                                 currSE->context = 1; //
1237                                 for choosing context model
1238                                 currSE->type =
1239                                     SE_LUM_AC_INTER;
1240                             }
1241                         }
1242                     }
1243                 }
1244             }
1245         }
1246     }
1247 }
1248

```

```

1229      /* choose the appropriate data
1230      partition */
1231      if (img->type != B_IMG)
          dataPart = &(currSlice->
          partArr[partMap[currSE->type
          ]]);
1232      else
1233          dataPart = &(currSlice->
          partArr[partMap[SE_BFRAME]]);
1234
1235      dataPart->writeSyntaxElement(
          currSE, img, inp, dataPart);
1236      bitCount[BITS_COEFF_Y_MB] += currSE
          ->len;
1237      #if TRACE
1238      sprintf(currSE->tracestring,
          "Luma sng(%2d) level =%3d run
          =%2d", k, level, run);
1239      #endif
1240
1241      /* proceed to next SE */
1242      currSE++;
1243      currMB->currSEnr++;
1244
1245      if (level!=0)
1246      {
1247          loopb[img->block_x+i+1][img->
          block_y+j+1]=max(loopb[img->
          block_x+i+1][img->block_y+j+1
          ],2);
1248
1249          loopb[img->block_x+i][img
          ->block_y+j+1]=max(loopb[img->
          block_x+i][img->block_y+j+
          1],1);
1250          loopb[img->block_x+i+1][img->
          block_y+j]=max(loopb[img->
          block_x+i+1][img->block_y+j
          ],1);
1251          loopb[img->block_x+i+2][img->
          block_y+j+1]=max(loopb[img->
          block_x+i+2][img->block_y+j+1
          ],1);
1252          loopb[img->block_x+i+1][img->
          block_y+j+2]=max(loopb[img->
          block_x+i+1][img->block_y+j+2
          ],1);
1253      }
1254      }
1255      }
1256      }
1257      }
1258      }
1259      }
1260      }
1261      }
1262      else /* 16x16 based intra modes */
1263      {
1264          /* DC coeffs */
1265          level=1; /* get inside loop */
1266          for (k=0;k<=16 && level !=0;k++)
1267          {
1268              level = currSE->value1 = img->cof[0][0][k][0][1]; // level
1269              run = currSE->value2 = img->cof[0][0][k][1][1]; // run
1270
1271              if (inp->symbol_mode == UVLC)

```

```

1272         currSE->mapping = levrn_lininfo_inter;
1273     else
1274     {
1275         currSE->context = 3; // for choosing context model
1276         currSE->writing = writeRunLevel2Buffer_CABAC;
1277     }
1278     currSE->type = SE_LUM_DC_INTRA; /* element is of type
DC */

1279
1280     /* choose the appropriate data partition */
1281     if (img->type != B_IMG)
1282         dataPart = &(currSlice->partArr[partMap[currSE->type
]])];
1283     else
1284         dataPart = &(currSlice->partArr[partMap[SE_BFRAME]]);
1285
1286     dataPart->writeSyntaxElement( currSE, img, inp, dataPart
);
1287     bitCount[BITS_COEFF_Y_MB]+=currSE->len;
1288     #if TRACE
1289     sprintf(currSE->tracestring, "DC luma 16x16 sng(%2d)
level =%3d run =%2d", k, level, run);
1290     #endif
1291
1292     /* proceed to next SE */
1293     currSE++;
1294     currMB->currSEnr++;
1295 }
1296 /* AC coeffs */
1297 if (img->kac==1)
1298 {
1299     for (mb_y=0; mb_y < 4; mb_y += 2)
1300     {
1301         for (mb_x=0; mb_x < 4; mb_x += 2)
1302         {
1303             for (j=mb_y; j < mb_y+2; j++)
1304             {
1305                 for (i=mb_x; i < mb_x+2; i++)
1306                 {
1307                     level=1; /* get inside loop */
1308                     for (k=0;k<16 && level !=0;k++)
1309                     {
1310                         level = currSE->value1 = img->cof[i][j
]][k][0][SINGLE_SCAN]; // level
1311                         run = currSE->value2 = img->cof[i][j][
k][1][SINGLE_SCAN]; // run
1312
1313                         if (inp->symbol_mode == UVLC)
1314                             currSE->mapping =
levrn_lininfo_inter;
1315                     else
1316                     {
1317                         currSE->context = 4; // for
choosing context model
1318                         currSE->writing =
writeRunLevel2Buffer_CABAC;
1319                     }
1320                     currSE->type = SE_LUM_AC_INTRA; /*
element is of type AC */
1321
1322                     /* choose the appropriate data
partition */
1323                     if (img->type != B_IMG)
1324                         dataPart = &(currSlice->partArr[
partMap[currSE->type]]);
1325

```

```

1326         else
1327             dataPart = &(currSlice->partArr[
                partMap[SE_BFRAME]]);
1328
1329             dataPart->writeSyntaxElement( currSE
                , img, inp, dataPart);
1330             bitCount[BITS_COEFF_Y_MB]+=currSE->len;
1331             #if TRACE
1332                 sprintf(currSE->tracestring, "AC luma
                    16x16 sng(%2d) level =%3d run =%2d", k
                    , level, run);
1333             #endif
1334             /* proceed to next SE */
1335             currSE++;
1336             currMB->currSEnr++;
1337         }
1338     }
1339 }
1340 }
1341 }
1342 }
1343 }
1344
1345 /* Bits for chroma 2x2 DC transform coefficients */
1346 m2=img->mb_x*2;
1347 jg2=img->mb_y*2;
1348
1349 if (cbp > 15) /* check if any chroma bits in coded block pattern
    is set */
1350 {
1351     for (uv=0; uv < 2; uv++)
1352     {
1353         level=1;
1354         for (k=0; k < 5 && level != 0; ++k)
1355         {
1356
1357             level = currSE->value1 = img->cofu[k][0][uv]; // level
1358             run = currSE->value2 = img->cofu[k][1][uv]; // run
1359
1360             if (inp->symbol_mode == UVLC)
1361                 currSE->mapping = levrn_linfo_c2x2;
1362             else
1363                 currSE->writing = writeRunLevel2Buffer_CABAC;
1364
1365             if ( img->imod == INTRA_MB_OLD || img->imod ==
                INTRA_MB_NEW)
1366             {
1367                 currSE->context = 6; // for choosing context model
1368                 currSE->type = SE_CHR_DC_INTRA;
1369             }
1370             else
1371             {
1372                 currSE->context = 5; // for choosing context model
1373                 currSE->type = SE_CHR_DC_INTER;
1374             }
1375
1376             /* choose the appropriate data partition */
1377             if (img->type != B_IMG)
1378                 dataPart = &(currSlice->partArr[partMap[currSE->
                    type]]);
1379             else
1380                 dataPart = &(currSlice->partArr[partMap[SE_BFRAME
                    ]]);
1381
1382             dataPart->writeSyntaxElement( currSE, img, inp,

```

```

dataPart);
bitCount[BITS_COEFF_UV_MB]+=currSE->len;
1383
1384     #if TRACE
1385         sprintf(currSE->tracestring, "2x2 DC Chroma %2d:
level =%3d run =%2d",k, level, run);

1386     #endif
1387
1388     /* proceed to next SE */
1389     currSE++;
1390     currMB->currSEnr++;
1391
1392
1393     if (level != 0)/* fix from ver 4.1 */
1394     {
1395         for (j=0;j<2;j++)
1396         {
1397             for (i=0;i<2;i++)
1398             {
1399                 loopc[m2+i+1][jg2+j+1]=max(loopc[m2+i+1][
jg2+j+1],2);
1400             }
1401         }
1402
1403         for (i=0;i<2;i++)
1404         {
1405             loopc[m2+i+1][jg2      ]=max(loopc[m2+i+1][
jg2      ],1);
1406             loopc[m2+i+1][jg2+3  ]=max(loopc[m2+i+1][jg2+
3  ],1);
1407             loopc[m2      ][jg2+i+1]=max(loopc[m2
][jg2+i+1],1);
1408             loopc[m2+3  ][jg2+i+1]=max(loopc[m2+3  ][jg2+i
+1],1);
1409         }
1410     }
1411 }
1412 }
1413 }
1414
1415 /* Bits for chroma AC-coeffs. */
1416 if (cbp >> 4 == 2) /* check if chroma bits in coded block pattern
= 10b */
1417 {
1418     for (mb_y=4; mb_y < 6; mb_y += 2)
1419     {
1420         for (mb_x=0; mb_x < 4; mb_x += 2)
1421         {
1422             for (j=mb_y; j < mb_y+2; j++)
1423             {
1424                 jj=j/2;
1425                 j1=j-4;
1426                 for (i=mb_x; i < mb_x+2; i++)
1427                 {
1428                     ii=i/2;
1429                     i1=i%2;
1430                     level=1;
1431                     for (k=0; k < 16 && level != 0; k++)
1432                     {
1433                         level = currSE->value1 = img->cof[i][j][k
][0][0]; // level
1434                         run = currSE->value2 = img->cof[i][j][k][1
][0]; // run
1435
1436                         if (inp->symbol_mode == UVLC)
1437                             currSE->mapping = levrunc_linfo_inter;
1438                         else

```

```

1439         currSE->writing =
1440         writeRunLevel2Buffer_CABAC;
1441
1442         if ( img->imod == INTRA_MB_OLD || img->
1443             imod == INTRA_MB_NEW)
1444         {
1445             currSE->context = 8; // for choosing
1446             context model
1447             currSE->type = SE_CHR_AC_INTRA;
1448         }
1449         else
1450         {
1451             currSE->context = 7; // for choosing
1452             context model
1453             currSE->type = SE_CHR_AC_INTER;
1454         }
1455         /* choose the appropriate data partition
1456         */
1457         if (img->type != B_IMG)
1458             dataPart = &(currSlice->partArr[
1459                 partMap[currSE->type]]);
1460         else
1461             dataPart = &(currSlice->partArr[
1462                 partMap[SE_BFRAME]]);
1463
1464         dataPart->writeSyntaxElement( currSE,
1465             img, inp, dataPart);
1466         bitCount[BITS_COEFF_UV_MB]+=currSE->len;
1467
1468         #if TRACE
1469         sprintf(currSE->tracestring, "AC Chroma
1470             %2d: level =%3d run =%2d",k, level, run);
1471
1472         #endif
1473
1474         /* proceed to next SE */
1475         currSE++;
1476         currMB->currSEnr++;
1477
1478         if (level != 0)
1479         {
1480             loopc[m2+i1+1][jg2+j1+1]=max(loopc[m2+
1481                 i1+1][jg2+j1+1],2);
1482
1483             loopc[m2+i1 ][jg2+j1+1]=max(loopc[m2+
1484                 i1 ][jg2+j1+1],1);
1485             loopc[m2+i1+1][jg2+j1 ]=max(loopc[m2+
1486                 i1+1][jg2+j1 ],1);
1487             loopc[m2+i1+2][jg2+j1+1]=max(loopc[m2+
1488                 i1+2][jg2+j1+1],1);
1489             loopc[m2+i1+1][jg2+j1+2]=max(loopc[m2+
1490                 i1+1][jg2+j1+2],1);
1491         }
1492     }
1493 }
1494 }
1495 }
1496 }
1497 }
1498 }
1499 }
1500 }
1501 }
1502 }
1503 }
1504 }
1505 }
1506 }
1507 }
1508 }
1509 }
1510 }
1511 }
1512 }
1513 }
1514 }
1515 }
1516 }
1517 }
1518 }
1519 }
1520 }
1521 }
1522 }
1523 }
1524 }
1525 }
1526 }
1527 }
1528 }
1529 }
1530 }
1531 }
1532 }
1533 }
1534 }
1535 }
1536 }
1537 }
1538 }
1539 }
1540 }
1541 }
1542 }
1543 }
1544 }
1545 }
1546 }
1547 }
1548 }
1549 }
1550 }
1551 }
1552 }
1553 }
1554 }
1555 }
1556 }
1557 }
1558 }
1559 }
1560 }
1561 }
1562 }
1563 }
1564 }
1565 }
1566 }
1567 }
1568 }
1569 }
1570 }
1571 }
1572 }
1573 }
1574 }
1575 }
1576 }
1577 }
1578 }
1579 }
1580 }
1581 }
1582 }
1583 }
1584 }
1585 }
1586 }
1587 }
1588 }
1589 }
1590 }
1591 }
1592 }
1593 }
1594 }
1595 }
1596 }
1597 }
1598 }
1599 }
1600 }
1601 }
1602 }
1603 }
1604 }
1605 }
1606 }
1607 }
1608 }
1609 }
1610 }
1611 }
1612 }
1613 }
1614 }
1615 }
1616 }
1617 }
1618 }
1619 }
1620 }
1621 }
1622 }
1623 }
1624 }
1625 }
1626 }
1627 }
1628 }
1629 }
1630 }
1631 }
1632 }
1633 }
1634 }
1635 }
1636 }
1637 }
1638 }
1639 }
1640 }
1641 }
1642 }
1643 }
1644 }
1645 }
1646 }
1647 }
1648 }
1649 }
1650 }
1651 }
1652 }
1653 }
1654 }
1655 }
1656 }
1657 }
1658 }
1659 }
1660 }
1661 }
1662 }
1663 }
1664 }
1665 }
1666 }
1667 }
1668 }
1669 }
1670 }
1671 }
1672 }
1673 }
1674 }
1675 }
1676 }
1677 }
1678 }
1679 }
1680 }
1681 }
1682 }
1683 }
1684 }
1685 }
1686 }
1687 }
1688 }
1689 }
1690 }
1691 }
1692 }
1693 }
1694 }
1695 }
1696 }
1697 }
1698 }
1699 }
1700 }
1701 }
1702 }
1703 }
1704 }
1705 }
1706 }
1707 }
1708 }
1709 }
1710 }
1711 }
1712 }
1713 }
1714 }
1715 }
1716 }
1717 }
1718 }
1719 }
1720 }
1721 }
1722 }
1723 }
1724 }
1725 }
1726 }
1727 }
1728 }
1729 }
1730 }
1731 }
1732 }
1733 }
1734 }
1735 }
1736 }
1737 }
1738 }
1739 }
1740 }
1741 }
1742 }
1743 }
1744 }
1745 }
1746 }
1747 }
1748 }
1749 }
1750 }
1751 }
1752 }
1753 }
1754 }
1755 }
1756 }
1757 }
1758 }
1759 }
1760 }
1761 }
1762 }
1763 }
1764 }
1765 }
1766 }
1767 }
1768 }
1769 }
1770 }
1771 }
1772 }
1773 }
1774 }
1775 }
1776 }
1777 }
1778 }
1779 }
1780 }
1781 }
1782 }
1783 }
1784 }
1785 }
1786 }
1787 }
1788 }
1789 }
1790 }
1791 }
1792 }
1793 }
1794 }
1795 }
1796 }
1797 }
1798 }
1799 }
1800 }
1801 }
1802 }
1803 }
1804 }
1805 }
1806 }
1807 }
1808 }
1809 }
1810 }
1811 }
1812 }
1813 }
1814 }
1815 }
1816 }
1817 }
1818 }
1819 }
1820 }
1821 }
1822 }
1823 }
1824 }
1825 }
1826 }
1827 }
1828 }
1829 }
1830 }
1831 }
1832 }
1833 }
1834 }
1835 }
1836 }
1837 }
1838 }
1839 }
1840 }
1841 }
1842 }
1843 }
1844 }
1845 }
1846 }
1847 }
1848 }
1849 }
1850 }
1851 }
1852 }
1853 }
1854 }
1855 }
1856 }
1857 }
1858 }
1859 }
1860 }
1861 }
1862 }
1863 }
1864 }
1865 }
1866 }
1867 }
1868 }
1869 }
1870 }
1871 }
1872 }
1873 }
1874 }
1875 }
1876 }
1877 }
1878 }
1879 }
1880 }
1881 }
1882 }
1883 }
1884 }
1885 }
1886 }
1887 }
1888 }
1889 }
1890 }
1891 }
1892 }
1893 }
1894 }
1895 }
1896 }
1897 }
1898 }
1899 }
1900 }
1901 }
1902 }
1903 }
1904 }
1905 }
1906 }
1907 }
1908 }
1909 }
1910 }
1911 }
1912 }
1913 }
1914 }
1915 }
1916 }
1917 }
1918 }
1919 }
1920 }
1921 }
1922 }
1923 }
1924 }
1925 }
1926 }
1927 }
1928 }
1929 }
1930 }
1931 }
1932 }
1933 }
1934 }
1935 }
1936 }
1937 }
1938 }
1939 }
1940 }
1941 }
1942 }
1943 }
1944 }
1945 }
1946 }
1947 }
1948 }
1949 }
1950 }
1951 }
1952 }
1953 }
1954 }
1955 }
1956 }
1957 }
1958 }
1959 }
1960 }
1961 }
1962 }
1963 }
1964 }
1965 }
1966 }
1967 }
1968 }
1969 }
1970 }
1971 }
1972 }
1973 }
1974 }
1975 }
1976 }
1977 }
1978 }
1979 }
1980 }
1981 }
1982 }
1983 }
1984 }
1985 }
1986 }
1987 }
1988 }
1989 }
1990 }
1991 }
1992 }
1993 }
1994 }
1995 }
1996 }
1997 }
1998 }
1999 }
2000 }

```

```

1  /*****
2  **
3  *   global.h for H.26L decoder.
4  *   Copyright (C) 1999   Telenor Satellite Services, Norway
5  *                           Ericsson Radio Systems, Sweden
6  *
7  *   Contacts:
8  *   Inge Lille-Langøy           <inge.lille-langoy@telenor.com>
9  *
10 *
11 *   Telenor Satellite Services
12 *   P.O.Box 6914 St.Olavs plass
13 *   N-0130 Oslo, Norway
14 *
15 *   Rickard Sjoberg             <rickard.sjoberg@era.ericsson.se>
16 *
17 *   Ericsson Radio Systems
18 *   KI/ERA/T/VV
19 *   164 80 Stockholm, Sweden
20 *****/
21 #ifndef _GLOBAL_H_
22 #define _GLOBAL_H_
23
24 #include <stdio.h>                                /* for FILE */
25
26 #define FALSE          0
27 #define TRUE           1
28
29 #define TRACE          1                          /* 0:Trace off
30 1:Trace on*/
31
32 typedef unsigned char  byte;                      /* 8 bit unsigned */
33
34 #define LEN_STARTCODE  31                          /* unique length of
35 startcode, defines new picture */
36
37 #define INTER_IMG_1     0
38 #define INTER_IMG_MULT  1
39 #define INTRA_IMG       2
40
41 // B pictures
42 #define B_IMG_1         3
43 #define B_IMG_MULT     4
44
45 /* coding of MBtypes */
46 #define INTRA_CODED_MB  0
47 #define INTER_CODED_MB  1
48
49 /* mb_mode constants */
50 #define COPY_MB         0                          /* just copy last MB,
51 no motion vectors */
52 #define M16x16_MB       1                          /* block sizes for
53 inter motion search */
54 #define M16x8_MB        2
55 #define M8x16_MB        3
56 #define M8x8_MB         4
57 #define M8x4_MB         5
58 #define M4x8_MB         6
59 #define M4x4_MB         7
60 #define INTRA_MB        8                          /* intra coded MB in
61 inter frame */
62
63 // B pictures : MB mode
64 #define INTRA_MB_B      16

```

```

60
61  /* imod constants */
62  #define INTRA_MB_OLD    0      /* new intra prediction mode in inter
frame */
63  #define INTRA_MB_NEW    1      /* 'old' intra prediction mode in
inter frame */
64  #define INTRA_MB_INTER  2      /* Intra MB in inter frame, use the
constants above */
65  // B pictures : imod
66  #define B_Forward        3
67  #define B_Backward       4
68  #define B_Bidirect       5
69  #define B_Direct         6
70
71  /* 4x4 intra prediction modes */
72  /*#define DC_PRED        0
73  #define VERT_PRED        1
74  #define HOR_PRED         2
75  #define DIAG_PRED_LR     3
76  #define DIAG_PRED_RL     4 */
77  /* 4x4 intra prediction modes */
78  #define DC_PRED          0
79  #define DIAG_PRED_RL     1
80  #define VERT_PRED        2
81  #define DIAG_PRED_LR_45  3
82  #define HOR_PRED         4
83  #define DIAG_PRED_LR     5
84
85  /* 16x16 intra prediction modes */
86  #define VERT_PRED_16     0
87  #define HOR_PRED_16      1
88  #define DC_PRED_16       2
89  #define PLANE_16         3
90
91
92  #define BLOCK_SIZE        4
93  #define MB_BLOCK_SIZE    16
94
95  #define MAX_MULT_PRED     5      /* max number of
reference frames */
96
97  /* QCIF format */
98  #define IMG_WIDTH         176      /* luma */
99  #define IMG_HEIGHT        144
100 #define IMG_WIDTH_CR      88      /* chroma */
101 #define IMG_HEIGHT_CR     72
102
103 /* image formats*/
104 #define SUB_QCIF           0      /* GH added picture formats*/
105 #define QCIF               1
106 #define CIF                2
107 #define CIF_4              3      /* GH added picture formats*/
108 #define CIF_16             4      /* GH added picture formats*/
109 #define CUSTOM             5      /* GH added picture formats*/
110
111
112 #define EOS_MASK           0x01      /* mask for end of
sequence (bit 1) */
113 #define ICIF_MASK         0x02      /* mask for image
format (bit 2) */
114 #define QP_MASK           0x7C      /* mask for
quant.parameter (bit 3->7) */
115 #define TR_MASK           0x7f80    /* mask for temporal
reference (bit 8->15) */
116
117 #define DECODING_OK        0

```

```

118 #define SEARCH_SYNC 1
119 #define PICTURE_DECODED 2
120
121 /* symbol types */
122 //<!pu>#define TYPE_HEADER 0
123 //<!pu>#define TYPE_MBHEADER 1
124 //<!pu>#define TYPE_MVD 2
125 //<!pu>#define TYPE_CBP 3
126 //<!pu>#define TYPE_2x2DC 4
127 //<!pu>#define TYPE_COEFF_Y 5
128 //<!pu>#define TYPE_COEFF_C 6
129 //<!pu>#define TYPE_EOS 7
130
131 #define MIN_PIX_VAL 0 /* lowest legal
values for 8 bit sample */
132 #define MAX_PIX_VAL 255 /* highest legal
values for 8 bit sample */
133
134 #ifndef WIN32
135 #define max(a, b) ((a) > (b) ? (a) : (b)) /* Macro returning max
value */
136 #define min(a, b) ((a) < (b) ? (a) : (b)) /* Macro returning min
value */
137 #endif
138 #define mmax(a, b) ((a) > (b) ? (a) : (b)) /* Macro returning
max value */
139 #define mmin(a, b) ((a) < (b) ? (a) : (b)) /* Macro returning
min value */
140
141 #define MVPRED_MEDIAN 0
142 #define MVPRED_L 1
143 #define MVPRED_U 2
144 #define MVPRED_UR 3
145
146 int refFrArr[72][88]; /* Array for
reference frames of each block */
147 int slice_numbers[396];
148
149 byte imgY[288][352]; /* array for the
decoded luma component */
150 byte imgY_pf[288][352]; /* Post filter luma
image */
151 byte imgUV[2][144][176]; /* array for the
chroma component */
152 byte imgUV_pf[2][144][176]; /* Post filter luma
image */
153
154 // B pictures
155 int Bframe_ctr;
156 byte prevP_tr, nextP_tr, P_interval;
157 int frame_no;
158
159 // byte mref[MAX_MULT_PRED][1152][1408]; /* 1/4 pix luma, not
used any more */
160 byte mref[MAX_MULT_PRED][288][352]; /* 1/1 pix luma for
direct interpolation */
161 byte mcef[MAX_MULT_PRED][2][288][352]; /* pix
chroma */
162
163 /* fix from ver 4.1 */
164 //byte loopb[91][75];
165 byte loopb[92][76];
166 /* fix from ver 4.1 */
167 //byte loopc[47][39];
168 byte loopc[48][40];
169

```

```

170  /* image parameters */
171  struct img_par
172  {
173      int number;                                /* frame
number                                          */
174      int current_mb_nr;
175      int current_slice_nr;
176      int format;                                /* frame format CIF or
QCIF                                          */
177      int tr;                                    /* temporal reference,
8 bit, wraps at 255 */
178      int frame_cycle;                          /* cycle through all
stored reference frames */
179      int qp;                                    /* quant for the
current frame                                */
180      int type;                                  /* image type
INTER/INTRA                                */
181      int width;
182      int height;
183      int width_cr;                             /* width chroma */
184      int height_cr;                            /* height chroma */
185      int mb_y;
186      int mb_x;
187      int block_y;
188      int pix_y;
189      int pix_x;
190      int pix_c_y;
191      int block_c_x;
192      int block_x;
193      int pix_c_x;
194      int mb_mode;
195      int imod;                                /* MB mode */
196      int mv[92][72][3];
197      int mpr[16][16];                         /* predicted block */
198
199      int m7[4][4];                             /* final 4x4
block                                          */
200      int cof[4][6][4][4];                     /* correction
coefficients from predicted */
201      int cofu[4];
202      int ipredmode[90][74];                   /* prediction type
*/ /* fix from ver 4.1 */
203      int quad[256];
204      int lpfilter[6][601];
205
206      byte li[8];                               /* 8 pix input to
loopfilter routine */
207      byte lu[8];                               /* 8 pix output from
loopfilter routine */
208
209      /* B pictures */
210      int fw_mv[92][72][3], bw_mv[92][72][3];
211      int fw_multiframe_no;
212      int fw_blc_size_h;
213      int fw_blc_size_v;
214      int bw_multiframe_no;
215      int bw_blc_size_h;
216      int bw_blc_size_v;
217  };
218
219  /* signal to noise ratio parameters */
220  struct snr_par
221  {
222      float snr_y;                              /* current Y
SNR                                          */
223      float snr_u;                              /* current U

```

```

        SNR                                */
224     float snr_v;                        /* current V
        SNR                                */
225     float snr_y1;                      /* SNR Y(dB) first
        frame                            */
226     float snr_ul;                      /* SNR U(dB) first
        frame                            */
227     float snr_v1;                      /* SNR V(dB) first
        frame                            */
228     float snr_ya;                      /* Average SNR Y(dB)
        remaining frames                */
229     float snr_ua;                      /* Average SNR U(dB)
        remaining frames                */
230     float snr_va;                      /* Average SNR V(dB)
        remaining frames                */
231 };
232
233 /* input parameters from configuration file */
234 struct inp_par
235 {
236     char infile[100];                  /* Telenor H.26L
        input                            */
237     char outfile[100];                /* Decoded YUV 4:2:0
        output                            */
238     char reffile[100];                /* Optional YUV 4:2:0
        reference file for SNR measurement */
239     int postfilter;                   /* postfilter
        (0=OFF,1=ON)                     */
240 };
241
242 #if TRACE
243 FILE *p_trace;
244 #endif
245
246 /* prototypes */
247 void get_symbol(char *tracestring, int *len, int *info, int type);
248 int decode_slice(struct img_par *img, struct inp_par *inp, int
        mb_nr, FILE *p_out);
249 void write_frame(struct img_par *img, int, FILE *p_out);
250
251 // B pictures
252 void write_prev_Pframe(struct img_par *img, FILE *p_out);
253 void copy_Pframe(struct img_par *img, int);
254 void oneforthpix_1(struct img_par *img);
255 void oneforthpix_2(struct img_par *img);
256 void initialize_Bframe( );
257 void decode_Bframe(struct img_par *img, int mb_nr, int stop_mb);
258 int mb_Bframe(struct img_par *img);
259
260 void itrans(struct img_par *img, int ioff, int joff, int i0, int j0);
261 int intraped(struct img_par *img, int ioff, int joff, int i4, int j4);
262 void onethirdpix(struct img_par *img);
263 void onethirdpix_separable(struct img_par *img);
264 int macroblock(struct img_par *img);
265 int linfo_mvd(int len, int info);
266 int image(struct img_par *img, struct inp_par *inp);
267 void linfo_levrun_inter(int len, int info, int *level, int *irun);
268 void linfo_levrun_intra(int len, int info, int *level, int *irun);
269 void linfo_levrun_c2x2(int len, int info, int *level, int *irun);
270 void find_snr(struct snr_par *snr, struct img_par *img, FILE *p_ref,
        int postfilter);
271 void init(struct img_par *img);
272 void loopfilter(struct img_par *img);
273 int loop(struct img_par *img, int ibl, int ibr, int longFilt, int
        chroma);
274 //int loop(struct img_par *img, int ibl, int ibr, int longFilt);

```

```
275 //void    loopfilter_new(struct img_par *img);
276
277
278 void    oneforthpix(struct img_par *img);
279 void    itrans_2(struct img_par *img);
280 int     intrapred_luma_2(struct img_par *img,int predmode);
281
282 int     GetVLCSymbol (byte buffer[],int totbitoffset,int *info);
283
284 int     inter_intra(struct img_par *img);
285
286 /* NAL function pointers */
287 void (*nal_init) ();
288 void (*nal_finit) ();
289 void (*nal_get_symbol) (int *len, int *info, int type);
290 int (*nal_startcode_follows) ();
291 int (*nal_find_startcode) (FILE *p_in, int *tr, int *qp, int *mb_nr,
292 int *format);
292 int (*nal_symbols_available) (int type);
293
294 /* NAL functions TML bitstream*/
295 void bits_tml_init();
296 void bits_tml_finit();
297 void bits_tml_get_symbol(int *len, int *info, int type);
298 int bits_tml_startcode_follows();
299 int bits_tml_find_startcode(FILE *p_in, int *tr, int *qp, int *mb_nr,
300 int *format);
300 int bits_tml_symbols_available (int type);
301
302 /* Slice functions Partition syntax */
303 void slice_tml_init();
304 void slice_tml_finit();
305 void slice_tml_get_symbol(int *len, int *info, int type);
306 int slice_tml_startcode_follows();
307 int slice_tml_find_startcode(FILE *p_in, int *tr, int *qp, int *mb_nr,
308 int *format);
308 int slice_tml_symbols_available (int type);
309
310 /* NAL functions Partition syntax */
311 void part_tml_init();
312 void part_tml_finit();
313 void part_tml_get_symbol(int *len, int *info, int type);
314 int part_tml_startcode_follows();
315 int part_tml_find_startcode(FILE *p_in, int *tr, int *qp, int *mb_nr,
316 int *format);
316 int part_tml_symbols_available (int type);
317
318 /* Functions for direct interpolation */
319 byte get_pixel(int ref_frame,int x_pos, int y_pos, struct img_par *img
320 );
320 byte get_pixel_nextP(int x_pos, int y_pos, struct img_par *img);
321 void copy2mref(struct img_par *img);
322 void copy2mref_1(struct img_par *img);
323 void copy2mref_2(struct img_par *img);
324
325 #endif
326
```

```

1  /*****
2  **
3  *   image.cpp for H.26L decoder.
4  *   Copyright (C) 1999   Telenor Satellite Services, Norway
5  *
6  *   Contacts:
7  *   Inge Lille-Langøy           <inge.lille-langoy@telenor.com>
8  *
9  *
10 *   Telenor Satellite Services
11 *   P.O.Box 6914 St.Olavs plass
12 *   N-0130 Oslo, Norway
13 *
14 *****/
15 #ifndef IMAGE_H
16 #define IMAGE_H
17
18 /* TAPs used in the oneforthpix() routine */
19 const int ONE_FOURTH_TAP[3][2] =
20 {
21     {20,20},
22     {-5,-4},
23     { 1, 0},
24 };
25 const byte FILTER_STR[32][4] = /* for new loopfilter */
26 {
27     {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},
28     {0,0,0,0},
29     {0,0,0,0},{0,0,0,1},{0,0,0,1},{0,0,1,1},{0,0,1,1},{0,0,1,1},{0,1,1,1},
30     {0,1,1,1},
31     {0,1,1,1},{0,1,1,2},{0,1,1,2},{0,1,1,2},{0,1,1,2},{0,1,2,3},{0,1,2,3},
32     {0,1,2,3},
33     {0,1,2,4},{0,2,3,4},{0,2,3,5},{0,2,3,5},{0,2,3,5},{0,2,4,7},{0,3,5,8},
34     {0,3,5,9},
35 };
36 const byte LIM[32] =
37 {
38     7,  8,  9, 10, 11, 12, 14, 16,
39     18, 20, 22, 25, 28, 31, 35, 39,
40     44, 49, 55, 62, 69, 78, 88, 98,
41     110,124,139,156,175,197,221,248
42 };
43
44 /* convert from H.263 QP to H.26L quant given by: quant=pow(2,QP/6) */
45 const int QP2QUANT[32]=
46 {
47     1, 1, 1, 1, 2, 2, 2, 2,
48     3, 3, 3, 4, 4, 4, 5, 6,
49     6, 7, 8, 9,10,11,13,14,
50     16,17,20,23,25,29,32,36
51 };
52
53 int two[6]   = {64,-320,3328,1280,-320,64};
54 int three[6] = {128,-640,2560,2560,-640,128};
55
56 int five[6][6] = {{4,-20,80,80,-20,4},
57                  {-20,100,-400,-400,100,-20},
58                  {80,-400,1600,1600,-400,80},
59                  {80,-400,1600,1600,-400,80},
60                  {-20,100,-400,-400,100,-20},
61                  {4,-20,80,80,-20,4}};
62
63 int six[6][6] = {{1,-5,52,20,-5,1},

```

```
61         {-5,25,-260,-100,25,-5},
62         {52,-260,2704,1040,-260,52},
63         {20,-100,1040,400,-100,20},
64         {-5,25,-260,-100,25,-5},
65         {1,-5,52,20,-5,1}
66     };
67     int seven[6][6] = {{2,-10,40,40,-10,2},
68         {-10,50,-200,-200,50,-10},
69         {104,-520,2080,2080,-520,104},
70         {40,-200,800,800,-200,40},
71         {-10,50,-200,-200,50,-10},
72         {2,-10,40,40,-10,2}
73     };
74
75 #endif
76
```

```

1  /*!
2  *   \file
3  *       Image.c
4  *   \brief
5  *       Decode a Slice
6  *
7  *   \author
8  *       Main contributors (see contributors.h for copyright,
9  *       address and affiliation details)
10  *
11  *       Inge Lille-Langøy
12  *       <inge.lille-langoy@telenor.com>
13  *       Rickard Sjoberg
14  *       <rickard.sjoberg@era.ericsson.se>
15  *       Jani Lainema
16  *       <jani.lainema@nokia.com>
17  *       Sebastian Purreiter
18  *       <sebastian.purreiter@mch.siemens.de>
19  *       Byeong-Moon Jeon
20  *       <jeonbm@lge.com>
21  *       Thomas Wedi
22  *       <wedi@tnt.uni-hannover.de>
23  */
24
25 #include "contributors.h"
26
27 #include <math.h>
28 #include <stdio.h>
29 #include <stdlib.h>
30
31 #include "global.h"
32 #include "elements.h"
33 #include "image.h"
34
35 /*!
36 *   \fn      decode_slice()
37 *   \brief   decode one slice
38 */
39 int decode_slice(
40     struct img_par *img, /*!< pointer to image parameters */
41     struct inp_par *inp, /*!< pointer to configuration parameters
42     read from the config file */
43     int mb_nr,           /*!< macroblock number from where to
44     start decoding */
45     FILE *p_out)
46 {
47     int len,info;
48     int scale_factor;
49     int i,j;
50     int mb_width, stop_mb;
51     static int first_P = TRUE;
52
53     if (mb_nr < 0)
54     {
55         stop_mb = mb_nr;
56         mb_nr = 0;
57     }
58
59     if(mb_nr == 0)
60     {
61 #if TRACE
62         printf ("DecodeSlice, found mb_nr == 0, start decoding
63         picture header\n");
64 #endif
65
66         get_symbol("Image Type",&len, &info, SE_PTYPE);
67         img->type = (int) pow(2,(len/2))+info-1; /* find image
68         type INTRA/INTER*/
69     }
70 }

```

```

59
60     if(img->type==INTRA_IMG) // I picture
61         prevP_tr=img->tr;
62     else if(img->type == INTER_IMG_1 || img->type ==
63         INTER_IMG_MULT) // P pictures
64         nextP_tr=img->tr;
65
66     if(img->type == INTER_IMG_1 || img->type == INTER_IMG_MULT)
67     {
68         if(first_P) // first P picture
69         {
70             first_P = FALSE;
71             P_interval=nextP_tr-prevP_tr;
72         }
73         else // all other P pictures
74         {
75             write_prev_Pframe(img, p_out); // imgY_prev,
76             imgUV_prev -> file // oneforthpix_2(img); // mref_P, mcef_p ->
77             mref[][][], mcef[][][]
78             copy2mref_2(img);
79         }
80     }
81
82     if (img->type > B_IMG_MULT)
83     {
84         set_ec_flag(SE_PTYPE);
85         img->type = INTER_IMG_1; /* concealed element */
86     }
87
88     if (img->format == QCIF)
89         scale_factor=1; /* QCIF */
90     else
91         scale_factor=2; /* CIF */
92
93     img->width      = IMG_WIDTH      * scale_factor; /* luma */
94     img->height     = IMG_HEIGHT     * scale_factor;
95
96     img->width_cr   = IMG_WIDTH_CR   * scale_factor; /* chroma */
97     img->height_cr  = IMG_HEIGHT_CR  * scale_factor;
98
99     for(i=0;i<img->width/BLOCK_SIZE+1;i++) /* set edge
100     to -1, indicate nothing to predict from */
101     {
102         img->ipredmode[i+1][0]=-1;
103     }
104     for(j=0;j<img->height/BLOCK_SIZE+1;j++)
105     {
106         img->ipredmode[0][j+1]=-1;
107     }
108
109     // I or P or B pictures
110     {
111         for (i=0;i<90;i++)
112         {
113             for (j=0;j<74;j++)
114             {
115                 loopb[i+1][j+1]=0;
116             }
117         }
118         for (i=0;i<46;i++)
119         {
120             for (j=0;j<38;j++)
121             {
122                 loopc[i+1][j+1]=0;
123             }
124         }
125     }
126 }

```

```

121     mb_width = img->width/16;
122     img->mb_x = mb_nr%mb_width;
123     img->mb_y = mb_nr/mb_width;
124
125     // I or P pictures
126     if(img->type <= INTRA_IMG)
127     {
128         for (; img->mb_y < img->height/MB_BLOCK_SIZE;img->mb_y++)
129         {
130             img->block_y = img->mb_y * BLOCK_SIZE;           /* vertical
131             luma block position */
132             img->pix_c_y=img->mb_y *MB_BLOCK_SIZE/2;         /* vertical
133             chroma macroblock position */
134             img->pix_y=img->mb_y *MB_BLOCK_SIZE;             /* vertical
135             luma macroblock position */
136
137             for (; img->mb_x < img->width/MB_BLOCK_SIZE;img->mb_x++)
138             {
139                 img->block_x=img->mb_x*BLOCK_SIZE;          /* luma
140                 block */
141                 img->pix_c_x=img->mb_x*MB_BLOCK_SIZE/2;
142                 img->pix_x=img->mb_x *MB_BLOCK_SIZE;         /*
143                 horizontal luma macroblock position */
144
145                 if(nal_startcode_follows() || (stop_mb++ == 0))
146                     return DECODING_OK;
147
148                 img->current_mb_nr = img->mb_y*img->width/
149                 MB_BLOCK_SIZE+img->mb_x;
150                 slice_numbers[img->current_mb_nr] = img->
151                 current_slice_nr;
152
153                 #if TRACE
154                 fprintf(p_trace, "\n***** Pic: %i (I/P) MB: %i
155                 Slice: %i *****\n\n", img->tr, img->current_mb_nr
156                 , slice_numbers[img->current_mb_nr]);
157
158                 #endif
159
160                 if(macroblock(img)==SEARCH_SYNC)             /* decode
161                 one frame */
162                     return SEARCH_SYNC;                     /* error in
163                     bitstream, look for synk word next frame */
164             }
165             img->mb_x = 0;
166         }
167         loopfilter(img);
168         if(img->number == 0)
169         {
170             // oneforthpix(img);
171             copy2mref(img);
172         }
173         else
174         {
175             // oneforthpix_1(img);
176             copy2mref_1(img);
177         }
178     }
179     else // B pictures
180     {
181         Bframe_ctr++;
182         initialize_Bframe();
183         decode_Bframe(img, mb_nr, stop_mb);
184         //g.b. post filter like loop filter
185         loopfilter(img);
186     }
187
188     return PICTURE_DECODED;

```

```

176     }
177
178     /*****
179     *
180     *   Name :           oneforthpix()
181     *
182     *   Description: Upsample 4 times and store in buffer for multiple
183     *                 reference frames
184     *
185     *   Remark:
186     *       Thomas Wedi: Not used any more. The pels are directly
187     *                   interpolated
188     *                   in the compensation module.
189     *****/
190 void oneforthpix(struct img_par *img)
191 {
192     int imgY_tmp[576][704];
193     int i,j,i2,j2;
194     int is,ie2,je2;
195     int uv;
196
197     img->frame_cycle=img->number % MAX_MULT_PRED;
198
199     for (j=0; j < img->height; j++)
200     {
201         for (i=0; i < img->width; i++)
202         {
203             i2=i*2;
204             is=(ONE_FOURTH_TAP[0][0]*(imgY[j][i]
205                                     ]+imgY[j][min(
206                                     img->width-1,i+1)))+
207             ONE_FOURTH_TAP[1][0]*(imgY[j][max(0,i-1)]+imgY[j][min(
208             img->width-1,i+2)))+
209             ONE_FOURTH_TAP[2][0]*(imgY[j][max(0,i-2)]+imgY[j][min(
210             img->width-1,i+3))+16)/32;
211             imgY_tmp[j][i2] =imgY[j][i];          /* 1/1 pix pos */
212             imgY_tmp[j][i2+1]=max(0,min(255,is)); /* 1/2 pix pos */
213         }
214     }
215     for (i=0; i < (img->width-1)*2+1; i++)
216     {
217         for (j=0; j < img->height; j++)
218         {
219             j2=j*4;
220             /* change for TML4, use 6 TAP vertical filter */
221             is=(ONE_FOURTH_TAP[0][0]*(imgY_tmp[j]
222                                     ][i]+imgY_tmp
223             [min(img->height-1,j+1)][i])+
224             ONE_FOURTH_TAP[1][0]*(imgY_tmp[max(0,j-1)][i]+imgY_tmp
225             [min(img->height-1,j+2)][i])+
226             ONE_FOURTH_TAP[2][0]*(imgY_tmp[max(0,j-2)][i]+imgY_tmp
227             [min(img->height-1,j+3)][i))+16)/32;
228
229             mref[img->frame_cycle][j2][i*2]=imgY_tmp[j][i]; /*
230             1/2 pix */
231             mref[img->frame_cycle][j2+2][i*2]=max(0,min(255,is)); /*
232             1/2 pix */
233         }
234     }
235
236     /* 1/4 pix */
237     /* luma */
238     ie2=(img->width-1)*4;
239     je2=(img->height-1)*4;

```

```

231     for (j=0;j<je2+1;j+=2)
232         for (i=0;i<ie2;i+=2)
233             mref[img->frame_cycle][j][i+1]=(mref[img->frame_cycle][j][
                i]+mref[img->frame_cycle][j][min(ie2,i+2)])/2;
234
235
236     for (i=0;i<ie2+1;i++)
237     {
238         for (j=0;j<je2;j+=2)
239         {
240             mref[img->frame_cycle][j+1][i]=(mref[img->frame_cycle][j][
                i]+mref[img->frame_cycle][min(je2,j+2)][i])/2;
241
242             /* "funny position" */
243             if( ((i&3) == 3)&&((j+1)&3) == 3))
244             {
245                 mref[img->frame_cycle][j+1][i] = (mref[img->
                    frame_cycle][j-2][i-3]+
246
247                                     mref[img->
                    frame_cycle][j+2][i-
                        3]+
248                                     mref[img->
                    frame_cycle][j-2][i+
                        1]+
249                                     mref[img->
                    frame_cycle][j+2][i+
                        1]+2)/4;
250             }
251         }
252     }
253     /* Chroma: */
254     for (uv=0; uv < 2; uv++)
255     {
256         for (i=0; i < img->width_cr; i++)
257         {
258             for (j=0; j < img->height_cr; j++)
259             {
260                 mcef[img->frame_cycle][uv][j][i]=imgUV[uv][j][i];/*
                    just copy 1/1 pix, interpolate "online" */
261             }
262         }
263     }
264 }
265
266 /*****
267 *
268 *   Name :           copy2mref()
269 *
270 *   Description: store reconstructed frames in multiple
271 *               reference buffer
272 *
273 *****/
274 void copy2mref(struct img_par *img)
275 {
276     int i,j,uv;
277
278     img->frame_cycle=img->number % MAX_MULT_PRED;
279     for (j=0; j < img->height; j++)
280         for (i=0; i < img->width; i++)
281             mref[img->frame_cycle][j][i]=imgY[j][i];
282
283     for (uv=0; uv < 2; uv++)
284         for (i=0; i < img->width_cr; i++)

```

```

285         for (j=0; j < img->height_cr; j++)
286             mcef[img->frame_cycle][uv][j][i]=imgUV[uv][j][i];/* just copy
                1/1 pix, interpolate "online" */
287
288     }
289
290
291
292     /*!
293     * \fn      find_snr()
294     * \brief   Find PSNR for all three components.Compare decoded frame
                with
295     *          the original sequence. Read inp->jumpd frames to reflect
                frame skipping.
296     */
297 void find_snr(
298     struct snr_par *snr,    /*!< pointer to snr parameters */
299     struct img_par *img,    /*!< pointer to image parameters */
300     FILE *p_ref,           /*!< filestream to reference YUV file */
301     int postfilter)        /*!< postfilterin on (=1) or off (=0) */
302 {
303     int i,j;
304     int diff_y,diff_u,diff_v;
305     int uv;
306     byte imgY_ref[288][352];
307     byte imgUV_ref[2][144][176];
308     int status;
309     byte diff;
310
311     if(img->type<=INTRA_IMG) // I, P pictures
312         frame_no=img->number*P_interval;
313     else // B pictures
314     {
315         diff=nextP_tr-img->tr;
316         frame_no=(img->number-1)*P_interval-diff;
317     }
318
319     rewind(p_ref);
320     status = fseek (p_ref, frame_no*img->height*img->width*3/2, 0);
321     if (status != 0) {
322         printf(" Error in seeking img->tr: %d\n", img->tr);
323         exit (-1);
324     }
325     for (j=0; j < img->height; j++)
326         for (i=0; i < img->width; i++)
327             imgY_ref[j][i]=fgetc(p_ref);
328     for (uv=0; uv < 2; uv++)
329         for (j=0; j < img->height_cr ; j++)
330             for (i=0; i < img->width_cr; i++)
331                 imgUV_ref[uv][j][i]=fgetc(p_ref);
332
333     img->quad[0]=0;
334     diff_y=0;
335     for (i=0; i < img->width; ++i)
336     {
337         for (j=0; j < img->height; ++j)
338         {
339             diff_y += img->quad[abs(imgY[j][i]-imgY_ref[j][i])];
340         }
341     }
342
343     /* Chroma */
344     diff_u=0;
345     diff_v=0;
346
347     for (i=0; i < img->width_cr; ++i)

```

```

348     {
349         for (j=0; j < img->height_cr; ++j)
350         {
351             diff_u += img->quad[abs(imgUV_ref[0][j][i]-imgUV[0][j][i]
352             diff_v += img->quad[abs(imgUV_ref[1][j][i]-imgUV[1][j][i]
353             ]]);
354         }
355     }
356     /* Collecting SNR statistics */
357     if (diff_y != 0)
358     {
359         snr->snr_y=(float) (10*log10(65025*(float) (img->width)*(img->
360         height)/(float)diff_y));          /* luma snr for current frame
361         */
362     }
363     if (diff_u != 0)
364     {
365         snr->snr_u=(float) (10*log10(65025*(float) (img->width)*(img->
366         height)/(float) (4*diff_u)));      /* chroma snr for current
367         frame */
368         snr->snr_v=(float) (10*log10(65025*(float) (img->width)*(img->
369         height)/(float) (4*diff_v)));      /* chroma snr for current
370         frame */
371     }
372     if (img->number == 0) /* first */
373     {
374         snr->snr_y1=(float) (10*log10(65025*(float) (img->width)*(img->
375         height)/(float)diff_y));          /* keep luma snr for first
376         frame */
377         snr->snr_u1=(float) (10*log10(65025*(float) (img->width)*(img->
378         height)/(float) (4*diff_u)));      /* keep chroma snr for first
379         frame */
380         snr->snr_v1=(float) (10*log10(65025*(float) (img->width)*(img->
381         height)/(float) (4*diff_v)));      /* keep chroma snr for first
382         frame */
383         snr->snr_ya=snr->snr_y1;
384         snr->snr_ua=snr->snr_u1;
385         snr->snr_va=snr->snr_v1;
386     }
387     else
388     {
389         if(img->type <= INTRA_IMG) // P pictures
390         {
391             snr->snr_ya=(float) (snr->snr_ya*(img->number+Bframe_ctr)+
392             snr->snr_y)/(img->number+Bframe_ctr+1);          /* average
393             snr chroma for all frames except first */
394             snr->snr_ua=(float) (snr->snr_ua*(img->number+Bframe_ctr)+
395             snr->snr_u)/(img->number+Bframe_ctr+1);          /* average
396             snr luma for all frames except first */
397             snr->snr_va=(float) (snr->snr_va*(img->number+Bframe_ctr)+
398             snr->snr_v)/(img->number+Bframe_ctr+1);          /* average
399             snr luma for all frames except first */
400         }
401         else // B pictures
402         {
403             snr->snr_ya=(float) (snr->snr_ya*(img->number+Bframe_ctr-1
404             )+snr->snr_y)/(img->number+Bframe_ctr);          /* average
405             snr chroma for all frames except first */
406             snr->snr_ua=(float) (snr->snr_ua*(img->number+Bframe_ctr-1
407             )+snr->snr_u)/(img->number+Bframe_ctr);          /* average
408             snr luma for all frames except first */
409             snr->snr_va=(float) (snr->snr_va*(img->number+Bframe_ctr-1

```

```

    )+snr->snr_v)/(img->number+Bframe_ctr);          /* average
    snr luma for all frames except first */

390     }
391 }
392 }
393
394
395 /*!
396 * \fn      loopfilter()
397 * \brief   Filter to reduce blocking artifacts. The filter strength
398 *          is QP dependent.
399 */
400
401 static unsigned char
402     overallActivity[256], qp_overallActivity = 255;
403 static int
404     beta;
405
406 void loopfilter(struct img_par *img)
407 {
408     static int MAP[32];
409     byte imgY_tmp[288][352];          /* temp luma image */
410     byte imgUV_tmp[2][144][176];     /* temp chroma image */
411
412     int x,y,y4,x4,k,uv,str1,str2;
413
414     if (qp_overallActivity != img->qp)
415     {
416         int
417             i, alpha,
418             ALPHA_TABLE[32] = {
419                 0, 0, 0, 0, 0, 0, 0, 0,
420                 3, 3, 3, 7, 7, 7, 12, 17,
421                 17, 22, 28, 34, 40, 47, 60, 67,
422                 82, 89, 112, 137, 153, 187, 213, 249
423             },
424             BETA_TABLE[32] = {
425                 0, 0, 0, 0, 0, 0, 0, 0,
426                 3, 3, 3, 4, 4, 4, 6, 6,
427                 6, 7, 8, 8, 9, 9, 10, 10,
428                 11, 11, 12, 12, 13, 13, 14, 14
429             };
430
431         for (i=0; i<32; i++)
432             MAP[i] = QP2QUANT[i]-1;
433
434         /*
435          * ALPHA(img->qp) = ceil(2.0 * MAP(img->qp) * ln(MAP(img->qp)))
436          */
437         alpha = ALPHA_TABLE[img->qp];
438
439         /*
440          * BETA(img->qp) = floor(4.0 * log(MAP[img->qp]) + 0.5)
441          */
442         beta = BETA_TABLE[img->qp];
443
444         for (k=0; k<256; k++)
445             if (2*k >= alpha+(alpha/2))
446                 overallActivity[k] = 1;
447             else if (2*k >= alpha)
448                 overallActivity[k] = 2;
449             else
450                 overallActivity[k] = 3;
451
452         qp_overallActivity = img->qp;
453

```

```
454     }
455
456     /* Luma hor */
457     for (y=0;y<img->height;y++)
458         for (x=0;x<img->width;x++)
459             imgY_tmp[y][x]=imgY[y][x];
460
461     for (y=0;y<img->height;y++)
462     {
463         y4=y/4;
464         for (x=4;x<img->width;x=x+4)
465         {
466             x4=x/4;
467
468             str1 = loopb[x4][y4+1];
469             str2 = loopb[x4+1][y4+1];
470
471             if((str1!=0)&&(str2!=0))
472             {
473                 /*
474                  * Copy one line of 4+4 reconstruction pels into the
475                  * filtering buffer.
476                  */
477                 for (k=0;k<8;k++)
478                     img->li[k]=imgY_tmp[y][x-4+k];
479
480                 /*
481                  * Check for longer filtering on macroblock border if either
482                  * block is intracoded.
483                  */
484                 if(!(x4%4) && (str1 == 3 || str2 == 3))
485                 {
486                     if(loop(img,str1,str2, 1, 0))
487                     {
488                         imgY_tmp[y][x-4+6] = img->lu[6];
489                         imgY[y][x-3] = img->lu[1];
490                         imgY[y][x+2] = img->lu[6];
491                     }
492                     else
493                         loop(img,str1,str2, 0, 0);
494
495                     /*
496                      * Place back the filtered values into the filtered
497                      * reconstruction buffer.
498                      */
499                     for (k=2;k<6;k++)
500                         imgY[y][x-4+k]=img->lu[k];
501                 }
502             }
503
504             /* Luma vert */
505             for (y=0;y<img->height;y++)
506                 for (x=0;x<img->width;x++)
507                     imgY_tmp[y][x]=imgY[y][x];
508
509             for (x=0;x<img->width;x++)
510             {
511                 x4=x/4;
512                 for (y=4;y<img->height;y=y+4)
513                 {
514                     y4=y/4;
515
516                     str1 = loopb[x4+1][y4];
517                     str2 = loopb[x4+1][y4+1];
```

```
517
518     if((str1 != 0) && (str2 != 0))
519     {
520         /*
521          * Copy one line of 4+4 reconstruction pels into the
522          * filtering buffer.
523          */
524         for(k=0;k<8;k++)
525             img->li[k]=imgY_tmp[y-4+k][x];
526
527         /*
528          * Check for longer filtering on macroblock border if either
529          * block is intracoded.
530          */
531         if(!(y%4) && (str1 == 3 || str2 == 3))
532         {
533             if(loop(img,str1,str2, 1, 0)){
534                 imgY_tmp[y-4+6][x] = img->lu[6];
535                 imgY[y-3][x] = img->lu[1];
536                 imgY[y+2][x] = img->lu[6];
537             }
538         }
539         else
540             loop(img,str1,str2, 0, 0);
541
542         /*
543          * Place back the filtered values into the filtered
544          * reconstruction buffer.
545          */
546         for (k=2;k<6;k++)
547             imgY[y-4+k][x]=img->lu[k];
548     }
549 }
550
551 /* chroma */
552
553 for (uv=0;uv<2;uv++)
554 {
555     /* horizontal chroma */
556     for(y=0;y<img->height_cr;y++)
557         for(x=0;x<img->width_cr;x++)
558             imgUV_tmp[uv][y][x]=imgUV[uv][y][x];
559
560     for(y=0;y<img->height_cr;y++)
561     {
562         y4=y/4;
563         for(x=4;x<img->width_cr;x=x+4)
564         {
565             x4=x/4;
566
567             str1 = loopc[x4][y4+1];
568             str2 = loopc[x4+1][y4+1];
569
570             if((str1 != 0) && (str2 != 0))
571             {
572                 /*
573                  * Copy one line of 4+4 reconstruction pels into the
574                  * filtering buffer.
575                  */
576                 for(k=0;k<8;k++)
577                     img->li[k]=imgUV_tmp[uv][y][x-4+k];
578
579                 /*
580                  * Check for longer filtering on macroblock border if
581                  * either block is intracoded.
582                  */

```

```

578     */
579     if(!(x4%2) && (loopb[2*x4][2*y4+1] == 3 || loopb[2*x4+1][2*
580     y4+1] == 3))
581         loop(img, str1, str2, 1, 1);
582     else
583         loop(img, str1, str2, 0, 1);
584
585     /*
586     * Place back the filtered values into the filtered
587     reconstruction buffer.
588     */
589     for (k=2; k<6; k++)
590         imgUV[uv][y][x-4+k] = img->lu[k];
591 }
592
593 /* vertical chroma */
594 for (y=0; y<img->height_cr; y++)
595     for (x=0; x<img->width_cr; x++)
596         imgUV_tmp[uv][y][x] = imgUV[uv][y][x];
597
598 for (x=0; x<img->width_cr; x++)
599 {
600     x4 = x/4;
601     for (y=4; y<img->height_cr; y=y+4)
602     {
603         y4 = y/4;
604
605         str1 = loopc[x4+1][y4];
606         str2 = loopc[x4+1][y4+1];
607
608         if((str1 != 0) && (str2 != 0))
609         {
610             /*
611             * Copy one line of 4+4 reconstruction pels into the
612             filtering buffer.
613             */
614             for (k=0; k<8; k++)
615                 img->li[k] = imgUV_tmp[uv][y-4+k][x];
616
617             /*
618             * Check for longer filtering on macroblock border if
619             either block is intracoded.
620             */
621             if(!(y4%2) && (loopb[2*x4+1][2*y4] == 3 || loopb[2*x4+1][2*
622             y4+1] == 3))
623                 loop(img, str1, str2, 1, 1);
624             else
625                 loop(img, str1, str2, 0, 1);
626
627             /*
628             * Place back the filtered values into the filtered
629             reconstruction buffer.
630             */
631             for (k=2; k<6; k++)
632                 imgUV[uv][y-4+k][x] = img->lu[k];
633         }
634     }
635 }
636
637 /*!
638 * \fn      loop()
639 * \brief   Filter 4 or 6 from 8 pix input

```

```
638     */
639
640     int loop(struct img_par *img, int ibl, int ibr, int longFilt, int
chroma)
641     {
642         int
643             delta, halfLim, dr, dl, i, truncLimLeft, truncLimRight,
644             diff, clip, clip_left, clip_right;
645
646         /* Limit the difference between filtered and nonfiltered based
647          * on QP and strength
648          */
649         clip_left  = FILTER_STR[img->qp][ibl];
650         clip_right = FILTER_STR[img->qp][ibr];
651
652         /* The step across the block boundaries */
653         delta  = abs(img->li[3]-img->li[4]);
654
655         /* Find overall activity parameter (n/2) */
656         halfLim = overallActivity[delta];
657
658         /* Truncate left limit to 2 for small stengths */
659         if (ibl <= 1)
660             truncLimLeft = (halfLim > 2) ? 2 : halfLim;
661         else
662             truncLimLeft = halfLim;
663
664         /* Truncate right limit to 2 for small stengths */
665         if (ibr <= 1)
666             truncLimRight = (halfLim > 2) ? 2 : halfLim;
667         else
668             truncLimRight = halfLim;
669
670         /* Find right activity parameter dr */
671         for (dr=1; dr<truncLimRight; dr++)
672         {
673             if (dr*abs(img->li[4]-img->li[4+dr]) > beta)
674                 break;
675         }
676
677         /* Find left activity parameter dl */
678         for (dl=1; dl<truncLimLeft; dl++)
679         {
680             if (dl*abs(img->li[3]-img->li[3-dl]) > beta)
681                 break;
682         }
683
684         if(dr < 2 || dl < 2)
685         {
686             /* no filtering when either one of activity params is below two */
687             img->lu[2] = img->li[2];
688             img->lu[3] = img->li[3];
689             img->lu[4] = img->li[4];
690             img->lu[5] = img->li[5];
691
692             return 0;
693         }
694
695         if(longFilt)
696         {
697             if(dr == 3 && dl == 3 &&
698                delta >= 2 && delta < img->qp/4)
699             {
700                 if(chroma)
701                 {
702                     img->lu[3] = (25*(img->li[1] + img->li[5]) + 26*(img->li[2] +
```

```

703     img->li[3] + img->li[4]) + 64) >> 7;
704     img->lu[2] = (25*(img->li[0] + img->li[4]) + 26*(img->li[1] +
img->li[2] + img->lu[3]) + 64) >> 7;
705     img->lu[4] = (25*(img->li[2] + img->li[6]) + 26*(img->li[3] +
img->li[4] + img->li[5]) + 64) >> 7;
706     img->lu[5] = (25*(img->li[3] + img->li[7]) + 26*(img->lu[4] +
img->li[5] + img->li[6]) + 64) >> 7;
707
708     return 0;
709 }
710 else
711 {
712     img->lu[3] = (25*(img->li[1] + img->li[5]) + 26*(img->li[2] +
img->li[3] + img->li[4]) + 64) >> 7;
713     img->lu[2] = (25*(img->li[0] + img->li[4]) + 26*(img->li[1] +
img->li[2] + img->lu[3]) + 64) >> 7;
714     img->lu[1] = (25*(img->li[1] + img->lu[3]) + 26*(img->li[0] +
img->li[1] + img->lu[2]) + 64) >> 7;
715     img->lu[4] = (25*(img->li[2] + img->lu[6]) + 26*(img->li[3] +
img->li[4] + img->li[5]) + 64) >> 7;
716     img->lu[5] = (25*(img->li[3] + img->li[7]) + 26*(img->lu[4] +
img->li[5] + img->li[6]) + 64) >> 7;
717     img->lu[6] = (25*(img->lu[4] + img->li[6]) + 26*(img->lu[5] +
img->li[6] + img->li[7]) + 64) >> 7;
718
719     return 1;
720 }
721 }
722
723 /* Filter pixels at the edge */
724 img->lu[4] = (21*(img->li[3] + img->li[5]) + 22*img->li[4] + 32) >>
6;
725 img->lu[3] = (21*(img->li[2] + img->li[4]) + 22*img->li[3] + 32) >>
6;
726
727 if(dr == 3)
728     img->lu[5] = (21*(img->lu[4] + img->li[6]) + 22*img->li[5] + 32)
>> 6;
729 else
730     img->lu[5] = img->li[5];
731
732 if(dl == 3)
733     img->lu[2] = (21*(img->li[1] + img->lu[3]) + 22*img->li[2] + 32)
>> 6;
734 else
735     img->lu[2] = img->li[2];
736
737 /* Clipping parameter depends on one table and left and right act
params */
738 clip = (clip_left + clip_right + dl + dr) / 2;
739
740 /* Pixels at the edge are clipped differently */
741 for (i=3; i<=4; i++)
742 {
743     diff = (int)img->lu[i] - (int)img->li[i];
744
745     if (diff)
746     {
747         if (diff > clip)
748             diff = clip;
749         else if (diff < -clip)
750             diff = -clip;
751
752         img->lu[i] = img->li[i] + diff;
753     }

```

```

754     }
755
756     /* pixel from left is clipped */
757     diff = (int)img->lu[2] - (int)img->li[2];
758
759     if (diff)
760     {
761         if (diff > clip_left)
762             diff = clip_left;
763         else if (diff < -clip_left)
764             diff = -clip_left;
765
766         img->lu[2] = img->li[2] + diff;
767     }
768
769     /* pixel from right is clipped */
770     diff = (int)img->lu[5] - (int)img->li[5];
771
772     if (diff)
773     {
774         if (diff > clip_right)
775             diff = clip_right;
776         else if (diff < -clip_right)
777             diff = -clip_right;
778
779         img->lu[5] = img->li[5] + diff;
780     }
781
782     return 0;
783 }
784
785 /*****
786  *
787  *   Name :           get_pixel()
788  *
789  *   Description: Direct interpolation of a specific subpel position
790  *
791  *   Author:         Thomas Wedi, 12.01.2001      <wedi@tnt.uni-hannover.de>
792  *
793  *   Remarks:        A further significant complexity reduction is possible
794  *                   if the direct interpolation is not performed on pixel
795  *                   basis
796  *                   but on block basis,
797  *
798  *
799  *****/
800 byte get_pixel(int ref_frame, int x_pos, int y_pos, struct img_par *img)
801 {
802     int dx=0, x=0;
803     int dy=0, y=0;
804     int maxold_x=0, maxold_y=0;
805
806     int result=0;
807     int pres_x=0;
808     int pres_y=0;
809
810     /*****/
811     /* applied filters */
812     /* */
813     /* X  2  3  2  X      */
814     /*  2  6  7  6      */
815     /*  3  7  5  7      */
816     /*  2  6  7  6      */

```

```

817      /* X          X      */
818      /*          */
819      /* X-fullpel positon */
820      /*          */
821      /*****/
822
823      dx = x_pos%4;
824      dy = y_pos%4;
825      maxold_x=img->width-1;
826      maxold_y=img->height-1;
827
828      if(dx==0&&dy==0) /* fullpel position */
829      {
830          pres_y=y_pos/4;
831          pres_x=x_pos/4;
832          result=mref[ref_frame][pres_y][pres_x];
833          return result;
834      }
835      else if(dx==3&&dy==3) /* funny position */
836      {
837          pres_y=y_pos/4;
838          pres_x=x_pos/4;
839          result=(mref[ref_frame][pres_y
840                                ]+[pres_x
841                                ]+
842                  mref[ref_frame][pres_y
843                                ]+[min(maxold_x,
844                                pres_x+1)]+
845                  mref[ref_frame][min(maxold_y,pres_y+1)][min(maxold_x,
846                                pres_x+1)]+
847                  mref[ref_frame][min(maxold_y,pres_y+1)][pres_x
848                                ]+2)/4;
849
850          return result;
851      }
852      else /* other positions */
853      {
854          if(x_pos==img->width*4-1)
855              dx=2;
856          if(y_pos==img->height*4-1)
857              dy=2;
858
859          if(dx==1&&dy==0)
860          {
861              pres_y=y_pos/4;
862              for(x=-2;x<4;x++)
863              {
864                  pres_x=max(0,min(maxold_x,x_pos/4+x));
865                  result+=mref[ref_frame][pres_y][pres_x]*two[x+2];
866              }
867          }
868          else if(dx==0&&dy==1)
869          {
870              pres_x=x_pos/4;
871              for(y=-2;y<4;y++)
872              {
873                  pres_y=max(0,min(maxold_y,y_pos/4+y));
874                  result+=mref[ref_frame][pres_y][pres_x]*two[y+2];
875              }
876          }
877          else if(dx==2&&dy==0)
878          {
879              pres_y=y_pos/4;
880              for(x=-2;x<4;x++)
881              {
882                  pres_x=max(0,min(maxold_x,x_pos/4+x));
883                  result+=mref[ref_frame][pres_y][pres_x]*three[x+2];
884              }
885          }
886      }

```

```
879     else if(dx==0&&dy==2)
880     {
881         pres_x=x_pos/4;
882         for(y=-2;y<4;y++)
883         {
884             pres_y=max(0,min(maxold_y,y_pos/4+y));
885             result+=mref[ref_frame][pres_y][pres_x]*three[y+2];
886         }
887     }
888     else if(dx==3&&dy==0)
889     {
890         pres_y=y_pos/4;
891         for(x=-2;x<4;x++)
892         {
893             pres_x=max(0,min(maxold_x,x_pos/4+x));
894             result+=mref[ref_frame][pres_y][pres_x]*two[3-x];    /* four */
895         }
896     }
897     else if(dx==0&&dy==3)
898     {
899         pres_x=x_pos/4;
900         for(y=-2;y<4;y++)
901         {
902             pres_y=max(0,min(maxold_y,y_pos/4+y));
903             result+=mref[ref_frame][pres_y][pres_x]*two[3-y];    /* four */
904         }
905     }
906     else if(dx==1&&dy==1)
907     {
908         for(y=-2;y<4;y++)
909         {
910             pres_y=max(0,min(maxold_y,y_pos/4+y));
911             for(x=-2;x<4;x++)
912             {
913                 pres_x=max(0,min(maxold_x,x_pos/4+x));
914                 result+=mref[ref_frame][pres_y][pres_x]*six[y+2][x+2];
915             }
916         }
917     }
918     else if(dx==2&&dy==1)
919     {
920         for(y=-2;y<4;y++)
921         {
922             pres_y=max(0,min(maxold_y,y_pos/4+y));
923             for(x=-2;x<4;x++)
924             {
925                 pres_x=max(0,min(maxold_x,x_pos/4+x));
926                 result+=mref[ref_frame][pres_y][pres_x]*seven[y+2][x+2];
927             }
928         }
929     }
930     else if(dx==1&&dy==2)
931     {
932         for(y=-2;y<4;y++)
933         {
934             pres_y=max(0,min(maxold_y,y_pos/4+y));
935             for(x=-2;x<4;x++)
936             {
937                 pres_x=max(0,min(maxold_x,x_pos/4+x));
938                 result+=mref[ref_frame][pres_y][pres_x]*seven[x+2][y+2]; /*
939                                     eight */
940             }
941         }
942     else if(dx==3&&dy==1)
943     {
```

```
944     for(y=-2;y<4;y++)
945     {
946         pres_y=max(0,min(maxold_y,y_pos/4+y));
947         for(x=-2;x<4;x++)
948         {
949             pres_x=max(0,min(maxold_x,x_pos/4+x));
950             result+=mref[ref_frame][pres_y][pres_x]*six[y+2][3-x]; /*
951             nine */
952         }
953     }
954     else if(dx==1&&dy==3)
955     {
956         for(y=-2;y<4;y++)
957         {
958             pres_y=max(0,min(maxold_y,y_pos/4+y));
959             for(x=-2;x<4;x++)
960             {
961                 pres_x=max(0,min(maxold_x,x_pos/4+x));
962                 result+=mref[ref_frame][pres_y][pres_x]*six[3-y][x+2]; /*
963                 ten */
964             }
965         }
966         else if(dx==2&&dy==2)
967         {
968             for(y=-2;y<4;y++)
969             {
970                 pres_y=max(0,min(maxold_y,y_pos/4+y));
971                 for(x=-2;x<4;x++)
972                 {
973                     pres_x=max(0,min(maxold_x,x_pos/4+x));
974                     result+=mref[ref_frame][pres_y][pres_x]*five[y+2][x+2];
975                 }
976             }
977         }
978         else if(dx==3&&dy==2)
979         {
980             for(y=-2;y<4;y++)
981             {
982                 pres_y=max(0,min(maxold_y,y_pos/4+y));
983                 for(x=-2;x<4;x++)
984                 {
985                     pres_x=max(0,min(maxold_x,x_pos/4+x));
986                     result+=mref[ref_frame][pres_y][pres_x]*seven[3-x][3-y]; /*
987                     eleven */
988                 }
989             }
990         else if(dx==2&&dy==3)
991         {
992             for(y=-2;y<4;y++)
993             {
994                 pres_y=max(0,min(maxold_y,y_pos/4+y));
995                 for(x=-2;x<4;x++)
996                 {
997                     pres_x=max(0,min(maxold_x,x_pos/4+x));
998                     result+=mref[ref_frame][pres_y][pres_x]*seven[3-y][x+2]; /*
999                     twelve */
1000                 }
1001             }
1002         else if(dx==3&&dy==3) /* not used if funny position is on */
1003         {
1004             for(y=-2;y<4;y++)
1005             {
```

```
1006         pres_y=max(0,min(maxold_y,y_pos/4+y));
1007         for(x=-2;x<4;x++)
1008         {
1009             pres_x=max(0,min(maxold_x,x_pos/4+x));
1010             result+=mref[ref_frame][pres_y][pres_x]*six[3-y][3-x]; /*
1011             thirteen */
1012         }
1013     }
1014 }
1015 }
1016
1017
1018     return max(0,min(255,(result+2048)/4096));
1019
1020 }
1021
1022
```

```

1  /*!
2  *   \file
3  *       Ldecod.c
4  *   \brief
5  *       TML decoder project main()
6  *   \author
7  *       Main contributors (see contributors.h for copyright,
address and affiliation details)
8  *
9  *       Inge Lille-Langøy           <inge.lille-langoy@telenor.com>
10  *       Rickard Sjoberg             <rickard.sjoberg@era.ericsson.se>
11  *       Stephan Wenger              <stewe@cs.tu-berlin.de>
12  *       Jani Lainema                <jani.lainema@nokia.com>
13  *       Sebastian Purreiter
<sebastian.purreiter@mch.siemens.de>
14  *       Byeong-Moon Jeon            <jeonbm@lge.com>
15  *
16  *   \note   tags are used for document system "doxygen"
17  *           available at
http://www.stack.nl/~dimitri/doxygen/index.html
18  *
19  *
20  *
21  *   \note
22  *       -Limitations:
23  *           Using different NAL's the assignment of partition-id to
containing
24  *           syntax elements may got lost, if this information is not
transmitted.
25  *           The same has to be stated for the partitionlength if
partitions are
26  *           merged by the NAL. \par
27  *
28  *           The presented solution in Q15-K-16 solves both of this
problems as the
29  *           departitioner parses the bitstream before decoding. Due
to syntax element
30  *           dependencies both, partition bounds and partitionlength
information can
31  *           be parsed by the departitioner. \par
32  *
33  *       -Handling partition information in external file:
34  *           As the TML is still a work in progress, it makes sense to
handle this
35  *           information for simplification in an external file, here
called partition
36  *           information file, which can be found by the extension .dp
extending the
37  *           original encoded H.26L bitstream. In this file
partition-ids followed by its
38  *           partitionlength is written. Instead of parsing the
bitstream we get the
39  *           partition information now out of this file.
40  *           This information is assumed to be never sent over
transmission channels
41  *           (simulation scenarios) as it's information we allways get
using a
42  *           "real" departitioner before decoding \par
43  *
44  *       -Extension of Interim File Format:
45  *           Therefore a convention has to be made within the interim
file format.
46  *           The underlying NAL has to take care of fulfilling these
conventions.
47  *           All partitions have to be bytealigned to be readable by
the decoder,

```

```

48      *           So if the NAL-encoder merges partitions, >>this is only
49      *           VLC structure of the H.26L bitstream<<, this bitaligned
50      *           structure has to be
51      *           broken up by the NAL-decoder. In this case the
52      *           NAL-decoder is responsible to
53      *           read the partitionlength information from the partition
54      *           information file.
55      *           Partitionlosses are signaled with a partition of zero
56      *           length containing no
57      *           syntax elements.
58      */
59
60      #include "contributors.h"
61
62      #include <stdlib.h>
63      #include <stdio.h>
64      #include <string.h>
65      #include <time.h>
66      #include <sys/timeb.h>
67
68      #include "global.h"
69      #include "elements.h"
70
71      #define TML          "5"
72      #define VERSION      "5.92"
73      #define LOGFILE      "log.dec"
74      #define DATADECFILE "data.dec"
75      #define TRACEFILE    "trace.dec"
76
77      /*!
78      *   \fn      main()
79      *   \brief   main function for TML decoder
80      */
81
82      int main(
83          int argc,
84          char **argv)
85      {
86          FILE *fd;                /* pointer to input configuration
87          file */
88          FILE *p_in;              /* pointer to input H.26L
89          bitstream file */
90          FILE *p_out;             /* pointer to output YUV file */
91          FILE *p_log;             /* pointer to output log file */
92          FILE *p_ref;             /* pointer to input original
93          reference YUV file */
94
95          char config_filename[100]; /* configuration filename */
96          char string[255];
97
98          time_t ltime1;           /* for time measurement */
99          time_t ltime2;
100
101      #ifdef WIN32
102          struct _timeb tstruct1;
103          struct _timeb tstruct2;
104          char timebuf[128];
105      #else
106          struct tm *l_time;
107          struct timeb tstruct1;
108          struct timeb tstruct2;
109          time_t now;
110      #endif

```

```

106
107     int tot_time=0;                /* time for total encoding
    session */
108     int tmp_time;                  /* time used by decoding the last
    frame */
109
110     struct inp_par    *inp;        /* input parameters from input
    configuration file */
111     struct snr_par    *snr;
112     struct img_par    *img;
113
114     int NAL_mode;
115     int mb_nr;
116
117     /* allocate memory for the structures */
118     inp = (struct inp_par *)calloc(1, sizeof(struct inp_par));
119     snr = (struct snr_par *)calloc(1, sizeof(struct snr_par));
120     img = (struct img_par *)calloc(1, sizeof(struct img_par));
121
122     if (argc != 2)
123     {
124         fprintf(stdout,"Usage: %s <config.dat> \n",argv[0]);
125         fprintf(stdout,"\t<config.dat> defines encoder parameters\n");
126         exit(-1);
127     }
128     strcpy(config_filename,argv[1]);
129     if((fd=fopen(config_filename,"r")) == NULL)                /* read the
    decoder configuration file */
130     {
131         fprintf(stdout,"Error: Control file %s not found\n",
            config_filename);
132         exit(0);
133     }
134
135     fscanf(fd,"%s",inp->infile);                /* YUV 4:2:2 input
    format */
136     fscanf(fd,"%s[^\\n]");
137
138     fscanf(fd,"%s",inp->outfile);                /* H.26L compressed
    output bitstream */
139     fscanf(fd,"%s[^\\n]");
140
141     fscanf(fd,"%s",inp->reffile);                /* reference file */
142     fscanf(fd,"%s[^\\n]");
143
144     fscanf(fd,"%d",&inp->postfilter);            /* not used */
145     fscanf(fd,"%s[^\\n]");
146
147     fscanf(fd,"%d",&(NAL_mode));                /* NAL mode */
148     fscanf(fd,"%s[^\\n]");
149
150     /*!
151     *   \note
152     *   NAL_mode=0 defines the conventional H.26L bitstream
153     *   all other NAL_modes are defined by assignSE2partition[][] in
154     *   elements.h
155     */
156     if (NAL_mode > 0)
157     {
158         PartitionMode = NAL_mode-1;
159         if (PartitionMode >= MAXPARTITIONMODES)
160         {
161             printf("NAL mode %i is not supported\n", NAL_mode);
162             exit(1);
163         }
164         NAL_mode = 1;

```

```

165     }
166
167     switch(NAL_mode)
168     {
169     case 0:
170         nal_init = bits_tml_init;
171         nal_finit = bits_tml_init;
172         nal_get_symbol = bits_tml_get_symbol;
173         nal_startcode_follows = bits_tml_startcode_follows;
174         nal_find_startcode = bits_tml_find_startcode;
175         nal_symbols_available = bits_tml_symbols_available;
176         break;
177     case 1:
178         nal_init = slice_tml_init;
179         nal_finit = slice_tml_init;
180         nal_get_symbol = slice_tml_get_symbol;
181         nal_startcode_follows = slice_tml_startcode_follows;
182         nal_find_startcode = slice_tml_find_startcode;
183         nal_symbols_available = slice_tml_symbols_available;
184         break;
185     case 2:
186         nal_init = part_tml_init;
187         nal_finit = part_tml_init;
188         nal_get_symbol = part_tml_get_symbol;
189         nal_startcode_follows = part_tml_startcode_follows;
190         nal_find_startcode = part_tml_find_startcode;
191         nal_symbols_available = part_tml_symbols_available;
192         break;
193     default:
194         printf("NAL mode %i is not supported\n",NAL_mode);
195         exit(1);
196     }
197
198     nal_init();
199
200     if ((p_in=fopen(inp->infile,"rb"))==0)
201     {
202         fprintf(stdout,"Input file %s does not exist \n",inp->infile);
203         exit(0);
204     }
205     if ((p_out=fopen(inp->outfile,"wb"))==0)
206     {
207         fprintf(stdout,"Error open file %s \n",inp->outfile);
208         exit(0);
209     }
210
211     #if TRACE
212     sprintf(string,"%s",TRACEFILE);
213     if ((p_trace=fopen(string,"w"))==0) /* append new
214         {
215             printf("Error open file %s!\n",string);
216             exit(0);
217         }
218     #endif
219
220     fprintf(stdout,
221             "-----\n");
221     fprintf(stdout," Decoder config file           : %s \n",
222             config_filename);
222     fprintf(stdout,
223             "-----\n");
223     fprintf(stdout," Input H.26L bitstream           : %s \n",
224             inp->infile);

```

```

224     fprintf(stdout, " Output decoded YUV 4:2:0           : %s \n",
        inp->outfile);
225     fprintf(stdout, " Output status file               : %s \n",
        LOGFILE);
226     if ((p_ref=fopen(inp->reffile,"rb"))==0)
227     {
228         fprintf(stdout, " Input reference file           : %s
        does not exist \n",inp->reffile);
229         fprintf(stdout, "                               SNR
        values are not available\n");
230     }
231     else
232         fprintf(stdout, " Input reference file           : %s
        \n",inp->reffile);
233     fprintf(stdout,
        "-----\n");
234
235
236     fprintf(stdout, "Frame    TR      QP   SnrY      SnrU      SnrV
        Time (ms) \n");
237
238     init(img);
239     img->number=0;
240     reset_ec_flags(); /*<pu> reset all error concealment flags */
241
242     /* B pictures */
243     Bframe_ctr=0;
244
245     while (nal_find_startcode(p_in, &img->tr, &img->qp, &mb_nr, &img->
        format) != 0) /* while not end of sequence, read next image to
        buffer */
246     {
247         int return_value;
248
249         if (ec_flag[SE_HEADER] == NO_EC)
250         {
251             if(mb_nr == 0)
252             {
253                 /* Picture start code */
254                 img->current_slice_nr = 0;
255             }
256             else
257             {
258                 /* Slice start code */
259                 img->current_slice_nr++;
260             }
261
262             #ifdef WIN32
263                 _ftime (&tstruct1); /* start time ms */
264             #else
265                 ftime (&tstruct1); /* start time ms */
266             #endif
267                 time( &lt;time1 ); /*
                start time s */
268
269                 return_value = decode_slice(img, inp, mb_nr, p_out);
270
271                 switch(return_value)
272                 {
273                     case SEARCH_SYNC:
274                         fprintf(stdout, "Biterror in frame %d\n",img->number);
275                         break;
276                     case DECODING_OK:
277                         break;
278                     case PICTURE_DECODED:

```

```

279         if (p_ref)
280             find_snr(snr,img,p_ref,inp->postfilter);          /*
                if ref sequence exist */
281
282     #ifdef WIN32
283         _ftime (&tstruct2);          /* end time ms */
284     #else
285         ftime (&tstruct2);          /* end time ms */
286     #endif
287
288     time( &ltime2 );          /*
        end time sec */
289     tmp_time=(ltime2*1000+tstruct2.millitm) - (ltime1*1000
        +tstruct1.millitm);
290     tot_time=tot_time + tmp_time;
291
292     if(img->type == INTRA_IMG) // I picture
293         fprintf(stdout,"%3d(I) %3d %5d %7.4f %7.4f %7.4f
        %5d\n",
294             frame_no, img->tr, img->qp,snr->snr_y,snr->
        snr_u,snr->snr_v,
295             tmp_time);
296     else if(img->type == INTER_IMG_1 || img->type ==
        INTER_IMG_MULT) // P pictures
297         fprintf(stdout,"%3d(P) %3d %5d %7.4f %7.4f %7.4f
        %5d\n",
298             frame_no, img->tr, img->qp,snr->snr_y,snr->
        snr_u,snr->snr_v,
299             tmp_time);
300     else // B pictures
301         fprintf(stdout,"%3d(B) %3d %5d %7.4f %7.4f %7.4f
        %5d\n",
302             frame_no, img->tr, img->qp,snr->snr_y,snr->
        snr_u,snr->snr_v,
303             tmp_time);
304
305     fflush(stdout);
306
307     if(img->type == INTER_IMG_1 || img->type ==
        INTER_IMG_MULT) // P pictres
308         copy_Pframe(img, inp->postfilter); //
        imgY-->imgY_prev, imgUV-->imgUV_prev
309     else // I or B pictures
310         write_frame(img,inp->postfilter,p_out);
311         /* write image to output YUV file */
312
313     if(img->type <= INTRA_IMG) // I or P pictures
314         img->number++;
315     break;
316 }
317 }
318 reset_ec_flags(); /*<pu> reset all error concealment flags */
319 // B PICTURE : save the last P picture
320 write_prev_Pframe(img, p_out);
321
322 fprintf(stdout,"----- Average SNR all frames
        -----\n");
323 fprintf(stdout," SNR Y(dB) : %5.2f\n",snr->snr_ya);
324 fprintf(stdout," SNR U(dB) : %5.2f\n",snr->snr_ua);
325 fprintf(stdout," SNR V(dB) : %5.2f\n",snr->snr_va);
326 fprintf(stdout," Total decoding time : %.3f sec \n",tot_time*0.001
        );
327 fprintf(stdout,
        "-----
        -----\n");
328 fprintf(stdout," Exit TML %s decoder, ver %s \n",TML,VERSION);

```

```

328
329     /* write to log file */
330
331     sprintf(string, "%s", LOGFILE);
332     if (fopen(string,"r")==0)                                /* check if file
333     exist */
334     {
335         if ((p_log=fopen(string,"a"))==0)
336         {
337             fprintf(stdout,"Error open file %s for appending\n",p_log);
338             exit(0);
339         }
340         else
341             /* Create
342             header to new file */
343         {
344             fprintf(p_log,"
345             -----
346             -----\\n");
347             fprintf(p_log,"| Decoder statistics. This file is made
348             first time, later runs are appended                |\\n");
349             fprintf(p_log,"
350             ----- \\n");
351             fprintf(p_log,"| Date   | Time   |      Sequence
352             |#Img|Format|SNRY  1|SNRU  1|SNRV  1|SNRY  N|SNRU  N|SNRV  N|\\n"
353             );
354             fprintf(p_log,"
355             -----
356             -----\\n");
357         }
358     }
359     else
360         p_log=fopen("log.dec","a");                            /* File
361         exist,just open for appending */
362
363     #ifdef WIN32
364         _strdate( timebuf );
365         fprintf(p_log,"| %1.5s |",timebuf );
366
367         _strtime( timebuf);
368         fprintf(p_log," % 1.5s |",timebuf);
369     #else
370         now = time ((time_t *) NULL);    /* Get the system time and put it
371         into 'now' as 'calender time' */
372         time (&now);
373         l_time = localtime (&now);
374         strftime (string, sizeof string, "%d-%b-%y", l_time);
375         fprintf(p_log,"| %1.5s |",string );
376
377         strftime (string, sizeof string, "%H:%M:%S", l_time);
378         fprintf(p_log,"| %1.5s |",string );
379     #endif
380
381     fprintf(p_log,"%20.20s|",inp->infile);
382
383     fprintf(p_log,"%3d |",img->number);
384
385     if(img->format==QCIF)
386         fprintf(p_log," QCIF |");
387     else
388         fprintf(p_log," CIF  |");
389
390     fprintf(p_log,"%6.3f|",snr->snr_y1);
391     fprintf(p_log,"%6.3f|",snr->snr_ul);
392     fprintf(p_log,"%6.3f|",snr->snr_v1);
393     fprintf(p_log,"%6.3f|",snr->snr_ya);

```

```

381     fprintf(p_log, "%6.3f|", snr->snr_ua);
382     fprintf(p_log, "%6.3f|\n", snr->snr_va);
383
384     fclose(p_log);
385
386     sprintf(string, "%s", DATADECFILE);
387     p_log=fopen(string, "a");
388
389     if(Bframe_ctr != 0) // B picture used
390     {
391         fprintf(p_log, "%3d %2d %2d %2.2f %2.2f %2.2f %5d "
392             "%2.2f %2.2f %2.2f %5d "
393             "%2.2f %2.2f %2.2f %5d %.3f\n",
394             img->number, 0, img->qp,
395             snr->snr_y1,
396             snr->snr_u1,
397             snr->snr_v1,
398             0,
399             0.0,
400             0.0,
401             0.0,
402             0,
403             snr->snr_ya,
404             snr->snr_ua,
405             snr->snr_va,
406             0,
407             (double)0.001*tot_time/(img->number+Bframe_ctr-1));
408     }
409     else
410     {
411         fprintf(p_log, "%3d %2d %2d %2.2f %2.2f %2.2f %5d "
412             "%2.2f %2.2f %2.2f %5d "
413             "%2.2f %2.2f %2.2f %5d %.3f\n",
414             img->number, 0, img->qp,
415             snr->snr_y1,
416             snr->snr_u1,
417             snr->snr_v1,
418             0,
419             0.0,
420             0.0,
421             0.0,
422             0,
423             snr->snr_ya,
424             snr->snr_ua,
425             snr->snr_va,
426             0,
427             (double)0.001*tot_time/img->number);
428     }
429
430     fclose(p_log);
431
432     return 0;
433 }
434
435
436 /*!
437 * \fn      init()
438 * \brief   Initilize some arrays
439 */
440 void init(struct img_par *img) /*!< image parameters */
441 {
442     int i;
443     /* initialize quad matrix used in snr routine */
444
445     for (i=0; i < 256; i++)
446     {

```

```
447         img->quad[i]=i*i; /* fix from TML 1, truncation removed */
448     }
449 }
450
451
452
```

```

1  /*!
2  *   \file
3  *       Macroblock.c
4  *   \brief
5  *       Decode a Macroblock
6  *   \author
7  *       Main contributors (see contributors.h for copyright,
address and affiliation details)
8  *       Inge Lille-Langøy
<inge.lille-langoy@telenor.com>
9  *       Rickard Sjoberg
<rickard.sjoberg@era.ericsson.se>
10  *       Jani Lainema                <jani.lainema@nokia.com>
11  *       Sebastian Purreiter
<sebastian.purreiter@mch.siemens.de>
12  *       Thomas Wedi                <wedi@tnt.uni-hannover.de>
13  */
14  #include "contributors.h"
15
16  #include <math.h>
17  #include <stdlib.h>
18  #include <stdio.h>
19  #include <assert.h>
20
21  #include "global.h"
22  #include "elements.h"
23  #include "macroblock.h"
24
25
26  /*!
27  *   \fn       ResetSlicePredictions()
28  *   \brief
29  */
30  void ResetSlicePredictions(struct img_par *img)
31  {
32      int x_pos = img->mb_x * 16;
33      int y_pos = img->mb_y * 16;
34      int mb_nr = img->current_mb_nr;
35      int mb_width = img->width/16;
36      /* change ipredmode for this block depending on slice */
37
38      if(x_pos >= 16)
39      {
40          if(slice_numbers[mb_nr] != slice_numbers[mb_nr-1])
41          {
42              img->ipredmode[x_pos/4][y_pos/4+1] = -1;
43              img->ipredmode[x_pos/4][y_pos/4+2] = -1;
44              img->ipredmode[x_pos/4][y_pos/4+3] = -1;
45              img->ipredmode[x_pos/4][y_pos/4+4] = -1;
46          }
47      }
48
49      if(y_pos >= 16)
50      {
51          if(slice_numbers[mb_nr] != slice_numbers[mb_nr-mb_width])
52          {
53              img->ipredmode[x_pos/4+1][y_pos/4] = -1;
54              img->ipredmode[x_pos/4+2][y_pos/4] = -1;
55              img->ipredmode[x_pos/4+3][y_pos/4] = -1;
56              img->ipredmode[x_pos/4+4][y_pos/4] = -1;
57          }
58      }
59  }
60
61
62  /*!

```

```

63  *  \fn      macroblock()
64  *
65  *  \brief   Decode one macroblock
66  *  \return  SEARCH_SYNC : error in bitstream
67  *          DECODING_OK  : no error was detected
68  */
69  int macroblock(
70      struct img_par *img)    /*!< image parameters (modified) */
71  {
72      int js[2][2];
73      int i,j,ii,jj,iii,jjj,i1,j1,ie,j4,i4,k,i0,pred_vec,j0;
74      int js0,js1,js2,js3,jf,mvDiffX,mvDiffY;
75      int coef_ctr,vec,cbp,uv,ll;
76      int scan_loop_ctr;
77      int level,run;
78      int ref_frame;
79      int len,info;
80      int predframe_no;        /* frame number which current MB is
predicted from */
81      int block_x,block_y;
82      int cbp_idx,dbl_ipred_word;
83      int vec1_x,vec1_y,vec2_x,vec2_y;
84      int scan_mode;
85
86      const int ICBPTAB[6] = {0,16,32,15,31,47};
87      int mod0;
88      int kmod;
89      int start_scan;
90      int ioff,joff;
91      int i3,j3;
92
93      int ii0,jj0,ii1,jj1,if1,jf1,if0,jf0;
94
95      int m2;
96      int jg2;
97
98      int interintra; /* distinction between Inter and Intra MBs */
99      int se;        /* syntax element type */
100
101
102      /* keep track of neighbouring macroblocks available for
prediction */
103      int mb_nr = img->current_mb_nr;
104      int mb_width = img->width/16;
105      int mb_available_up = (img->mb_y == 0) ? 0 : (slice_numbers[mb_nr]
== slice_numbers[mb_nr-mb_width]);
106      int mb_available_left = (img->mb_x == 0) ? 0 : (slice_numbers[
mb_nr] == slice_numbers[mb_nr-1]);
107      int mb_available_upleft = (img->mb_x == 0 || img->mb_y == 0) ? 0
: (slice_numbers[mb_nr] == slice_numbers[mb_nr-mb_width-1]);
108      int mb_available_upright = (img->mb_x >= mb_width-1 || img->mb_y
== 0) ? 0 : (slice_numbers[mb_nr] == slice_numbers[mb_nr-mb_width+
1]);
109
110      /* keep track of neighbouring blocks available for motion vector
prediction */
111      int block_available_up, block_available_left,
block_available_upright, block_available_upleft;
112      int mv_a, mv_b, mv_c, mv_d;
113      int mvPredType, rFrameL, rFrameU, rFrameUR;
114
115      ResetSlicePredictions(img);
116
117      if (img->type==INTER_IMG_1 || (img->type==INTER_IMG_MULT))
/* inter frame */
118      {

```

```

119     get_symbol("MB mode",&len,&info,SE_MBTYP);
120     img->mb_mode = (int)pow(2,(len/2))+info-1;          /* inter
prediction mode (block shape) */
121     if (img->mb_mode>32) /* 8 inter + 24 new intra */
122     {
123         /*!
124         * \note
125         *   Error concealment for MBTYPE:
126         *   Copy macroblock of last decoded picture
127         */
128         set_ec_flag(SE_MBTYP);
129         img->imod = INTRA_MB_INTER;
130         img->mb_mode = COPY_MB;
131     }
132     else /* no error concealment necessary */
133     {
134         if (img->mb_mode == INTRA_MB) /* 4x4 intra */
135             img->imod=INTRA_MB_OLD;
136         if (img->mb_mode > INTRA_MB) /* 16x16 intra */
137         {
138             img->imod=INTRA_MB_NEW;
139             mod0=img->mb_mode - INTRA_MB-1;
140             kmod=mod0 & 3;
141             cbp = ICBPTAB[mod0/4];
142         }
143         if (img->mb_mode < INTRA_MB) /* intra in inter frame */
144             img->imod=INTRA_MB_INTER;
145     }
146 }
147 else if ((img->type==INTRA_IMG))
148 {
149     get_symbol("Mb mode",&len,&info,SE_MBTYP);
150
151     img->mb_mode = (int)pow(2,(len/2))+info-1;
152     if (img->mb_mode>24)
153     { /* 24 new intra */
154         /*!
155         * \note
156         *   Error concealment for Intra macroblocks:
157         *   as for Intra images we do not have a COPY_MB type for
macroblocks, we
158         *   switch only for this macroblock to a INTER imod to
copy the information
159         *   from the last decoded MB
160         */
161         set_ec_flag(SE_MBTYP);
162         img->imod = INTRA_MB_INTER;
163         img->mb_mode = COPY_MB;
164     }
165     else /* no error concealment necessary */
166     {
167         if (img->mb_mode == 0)
168             img->imod=INTRA_MB_OLD; /* 4x4 intra */
169         else
170         {
171             img->imod=INTRA_MB_NEW; /* new */
172             mod0=img->mb_mode-1;
173             kmod=mod0 & 3;
174             cbp = ICBPTAB[mod0/4];
175         }
176     }
177 }
178 else
179 {
180     /* no error concealment if PTYPE is damaged */
181     printf("Wrong image type, exiting\n");

```

```

182         return SEARCH_SYNC;
183     }
184
185     img->mv[img->block_x+4][img->block_y][2]=img->number;
186
187     for (i=0;i<BLOCK_SIZE;i++)
188     {
189         /* reset vectors and pred. modes */
190         for(j=0;j<BLOCK_SIZE;j++)
191         {
192             img->mv[img->block_x+i+4][img->block_y+j][0] = 0;
193             img->mv[img->block_x+i+4][img->block_y+j][1] = 0;
194             img->ipredmode[img->block_x+i+1][img->block_y+j+1] = 0;
195         }
196     }
197     ref_frame = img->frame_cycle;
198     predframe_no=1;
199
200     /* Set the reference frame information for motion vector
201     prediction */
202     if (img->imod == INTRA_MB_OLD || img->imod == INTRA_MB_NEW)
203     {
204         for (j = 0;j < BLOCK_SIZE;j++)
205         {
206             for (i = 0;i < BLOCK_SIZE;i++)
207             {
208                 refFrArr[img->block_y+j][img->block_x+i] = -1;
209             }
210         }
211     }
212     else
213     {
214         for (j = 0;j < BLOCK_SIZE;j++)
215         {
216             for (i = 0;i < BLOCK_SIZE;i++)
217             {
218                 refFrArr[img->block_y+j][img->block_x+i] =
219                 predframe_no;
220             }
221         }
222     }
223
224     if (img->imod==INTRA_MB_INTER && img->mb_mode==COPY_MB) /*keep
225     last macroblock*/
226     {
227         // luma
228         for(j=0;j<MB_BLOCK_SIZE;j++)
229         {
230             jj=img->pix_y+j;
231             for(i=0;i<MB_BLOCK_SIZE;i++)
232             {
233                 ii=img->pix_x+i;
234                 imgY[jj][ii]=get_pixel(ref_frame,ii*4,jj*4,img);
235                 // imgY[jj][ii]=mref[ref_frame][jj*4][ii*4];
236             }
237         }
238
239         // chroma
240         for(uv=0;uv<2;uv++)
241         {
242             for(j=0;j<MB_BLOCK_SIZE/2;j++)
243             {
244                 jj=img->pix_c_y+j;
245                 for(i=0;i<MB_BLOCK_SIZE/2;i++)
246                 {
247                     ii=img->pix_c_x+i;
248                     imgUV[uv][jj][ii]=mcef[ref_frame][uv][jj][ii];

```

```

245     }
246     }
247 }
248
249 for (i=0;i<4;i++)
250 {
251     ii=img->block_x+i;
252     i3=ii/2;
253     for (j=0;j<4;j++)
254     {
255         jj=img->block_y+j;
256         j3=jj/2;
257
258         if (((img->mv[ii-1+4][jj][0]/4)!=0 || (img->mv[ii-1+4][
259             jj][1]/4!=0)) && ii > 0)
260         {
261             loopb[ii][jj+1]=max(loopb[ii][jj+1],1);
262             loopb[ii+1][jj+1]=max(loopb[ii+1][jj+1],1);
263             loopc[i3][j3+1]=max(loopc[i3][j3+1],1);
264             loopc[i3+1][j3+1]=max(loopc[i3+1][j3+1],1);
265         }
266         if (((img->mv[ii+4][jj-1][0]/4!=0 || (img->mv[ii+4][jj-
267             1][1]/4!=0)) && jj > 0)
268         {
269             loopb[ii+1][jj]=max(loopb[ii+1][jj],1);
270             loopb[ii+1][jj+1]=max(loopb[ii+1][jj+1],1);
271             loopc[i3+1][j3]=max(loopc[i3+1][j3],1);
272             loopc[i3+1][j3+1]=max(loopc[i3+1][j3+1],1);
273         }
274     }
275     return DECODING_OK;
276 }
277
278 if (img->imod==INTRA_MB_OLD) /* find two and two
279     intra prediction mode for a macroblock */
280 {
281     for(i=0;i<MB_BLOCK_SIZE/2;i++)
282     {
283         get_symbol("Intra mode", &len,&info,SE_INTRAPREDMODE);
284         dbl_ipred_word = (int) pow(2,(len/2))+info-1;
285         if (dbl_ipred_word>35)
286         {
287             /*!
288             * \note
289             * Error concealment for intra prediction mode,
290             * sets IntraPredictionMode to 0.
291             */
292             set_ec_flag(SE_INTRAPREDMODE);
293             get_concealed_element(&len, &info, SE_INTRAPREDMODE);
294         }
295
296         i1=img->block_x + 2*(i&0x01);
297         j1=img->block_y + i/2;
298
299         /* find intra prediction mode for two blocks */
300         img->ipredmode[i1+1][j1+1] = PRED_IPRED[img->ipredmode[i1+
301             1][j1+1]+1][img->ipredmode[i1][j1+1]+1][IPRED_ORDER[
302             dbl_ipred_word][0]];
303         img->ipredmode[i1+2][j1+1] = PRED_IPRED[img->ipredmode[i1+
304             2][j1+1]+1][img->ipredmode[i1+1][j1+1]+1][IPRED_ORDER[
305             dbl_ipred_word][1]];
306     }
307 }
308
309 if (img->imod==INTRA_MB_INTER) /* inter frame, read

```

```

vector data */
304 {
305     if(img->type==INTER_IMG_MULT)
306     {
307         get_symbol("Reference frame",&len,&info,SE_REFFRAME);
308         predframe_no=(int)pow(2,len/2)+info; /* find frame
number to copy blocks from (if mult.pred mode)*/
309         if (predframe_no > MAX_MULT_PRED)
310         {
311             /*!
312              * \note
313              * Error concealment for REFFRAME:
314              * predict from last decoded frame and set all MVD's
315              * and the CBP to zero
316              * \sa errorconcealment.c
317              */
318             set_ec_flag(SE_REFFRAME);
319             predframe_no = 1;
320         }
321         ref_frame=(img->frame_cycle+6-predframe_no) %
MAX_MULT_PRED;
322
323         /*!
324          * \note
325          * if frame lost occurs within MAX_MULT_PRED frames and
buffer of previous
326          * decoded pictures is still empty, set ref_frame to
last decoded
327          * picture to avoid prediction from unexistent frames.
328          */
329         if (ref_frame > img->number)
330         {
331             ref_frame = 0;
332         }
333
334         /* Update the reference frame information for motion
vector prediction */
335         for (j = 0;j < BLOCK_SIZE;j++)
336         {
337             for (i = 0;i < BLOCK_SIZE;i++)
338             {
339                 refFrArr[img->block_y+j][img->block_x+i] =
predframe_no;
340             }
341         }
342
343     }
344     ie=4-BLOCK_STEP[img->mb_mode][0];
345     for(j=0;j<4;j=j+BLOCK_STEP[img->mb_mode][1]) /* Y position
in 4-pel resolution */
346     {
347         block_available_up = mb_available_up || (j > 0);
348         j4=img->block_y+j;
349         for(i=0;i<4;i=i+BLOCK_STEP[img->mb_mode][0]) /* X
position in 4-pel resolution */
350         {
351             i4=img->block_x+i;
352
353             /* D B C */
354             /* A X */
355
356             /* 1 A, B, D are set to 0 if unavailable */
357             /* 2 If C is not available it is replaced by D */
358
359             block_available_left = mb_available_left || (i > 0);
360

```

```

361     if (j > 0)
362         block_available_upright = i != ie ? 1 : 0;
363     else if (i != ie)
364         block_available_upright = block_available_up;
365     else
366         block_available_upright = mb_available_upright;
367
368     if (i > 0)
369         block_available_upleft = j > 0 ? 1 :
370         block_available_up;
371     else if (j > 0)
372         block_available_upleft = block_available_left;
373     else
374         block_available_upleft = mb_available_upleft;
375
376     mvPredType = MVPRED_MEDIAN;
377
378     rFrameL    = block_available_left    ? refFrArr[j4][i4
379     -1]    : -1;
380     rFrameU    = block_available_up      ? refFrArr[j4-1][
381     i4]    : -1;
382     rFrameUR   = block_available_upright ? refFrArr[j4-1][
383     i4+BLOCK_STEP[img->mb_mode][0]] :
384     block_available_upleft ? refFrArr[j4-1][
385     i4-1] : -1;
386
387     /* Prediction if only one of the neighbors uses the
388     selected reference frame */
389
390     if(rFrameL == predframe_no && rFrameU != predframe_no
391     && rFrameUR != predframe_no)
392         mvPredType = MVPRED_L;
393     else if(rFrameL != predframe_no && rFrameU ==
394     predframe_no && rFrameUR != predframe_no)
395         mvPredType = MVPRED_U;
396     else if(rFrameL != predframe_no && rFrameU !=
397     predframe_no && rFrameUR == predframe_no)
398         mvPredType = MVPRED_UR;
399
400     /* Directional predictions */
401
402     else if(img->mb_mode == 3)
403     {
404         if(i == 0)
405         {
406             if(rFrameL == predframe_no)
407                 mvPredType = MVPRED_L;
408         }
409         else
410         {
411             if(rFrameUR == predframe_no)
412                 mvPredType = MVPRED_UR;
413         }
414     }
415     else if(img->mb_mode == 2)
416     {
417         if(j == 0)
418         {
419             if(rFrameU == predframe_no)
420                 mvPredType = MVPRED_U;
421         }
422         else
423         {
424             if(rFrameL == predframe_no)
425                 mvPredType = MVPRED_L;
426         }
427     }

```

```

418     }
419     else if(img->mb_mode == 5 && i == 2)
420         mvPredType = MVPRED_L;
421     else if(img->mb_mode == 6 && j == 2)
422         mvPredType = MVPRED_U;
423
424     for (k=0;k<2;k++) /* vector
prediction. find first koordinate set for pred */
425     {
426         mv_a = block_available_left ? img->mv[i4-1+
BLOCK_SIZE][j4][k] : 0;
427         mv_b = block_available_up ? img->mv[i4+
BLOCK_SIZE][j4-1][k] : 0;
428         mv_d = block_available_upleft ? img->mv[i4-1+
BLOCK_SIZE][j4-1][k] : 0;
429         mv_c = block_available_upright ? img->mv[i4+
BLOCK_SIZE][j4-1][k] :
mv_d;
430
431         switch (mvPredType)
432         {
433
434             case MVPRED_MEDIAN:
435
436                 if(!(block_available_upleft ||
block_available_up || block_available_upright))
437                     pred_vec = mv_a;
438                 else
439                     pred_vec =mv_a+mv_b+mv_c-min(mv_a,min(mv_b
,mv_c))-max(mv_a,max(mv_b,mv_c));
440
441                 break;
442
443             case MVPRED_L:
444                 pred_vec = mv_a;
445                 break;
446             case MVPRED_U:
447                 pred_vec = mv_b;
448                 break;
449             case MVPRED_UR:
450                 pred_vec = mv_c;
451                 break;
452             default:
453                 break;
454         }
455
456         get_symbol("MVD",&len,&info,SE_MVD);
457         if (len>17)
458         {
459             /*!
460              * \note
461              * Errorconcealment for motion vectors:
462              * set all MVD to zero as well as possible
CBP information
463              */
464             set_ec_flag(SE_MVD);
465             get_concealed_element(&len, &info, SE_MVD);
466         }
467         vec=linfo_mvd(len,info)+pred_vec; /*
find motion vector */
468         for(ii=0;ii<BLOCK_STEP[img->mb_mode][0];ii++)
469         {
470             for(jj=0;jj<BLOCK_STEP[img->mb_mode][1];jj++)
471             {
472                 img->mv[i4+ii+BLOCK_SIZE][j4+jj][k]=vec;
473             }

```

```

474         }
475     }
476 }
477 }
478 }
479
480
481 interintra = inter_intra(img); /* <pu> get information if MB is
INTRA or INTER coded */
482
483 /* read CBP if not new intra mode */
484 if (img->imod != INTRA_MB_NEW)
485 {
486     /* <pu> check for Inter or Intra CBP syntax element */
487     if ( interintra == INTRA_CODED_MB )
488         se = SE_CBP_INTRA;
489     else
490         se = SE_CBP_INTER;
491     get_symbol("CBP",&len,&info,se);
492
493     cbp_idx = (int) pow(2,(len/2))+info-1; /* find cbp index
in VLC table */
494     if (cbp_idx>47) /* illegal value,
exit to find next sync word */
495     {
496         /*!
497          * \note
498          * Error concealment for Coded Block Pattern:
499          * Set CBP to no texture information for all following
500          * CBP elements of either type INTER or INTRA
501          */
502         printf("Error in index n to array cbp\n");
503         set_ec_flag(se);
504         get_concealed_element(&len, &info, se);
505     }
506     cbp=NCBP[cbp_idx][img->imod/INTRA_MB_INTER];/* find coded
block pattern for intra or inter */
507 }
508
509 for (i=0;i<BLOCK_SIZE;i++)
510     for (j=0;j<BLOCK_SIZE;j++)
511         for(iii=0;iii<BLOCK_SIZE;iii++)
512             for(jjj=0;jjj<BLOCK_SIZE;jjj++)
513                 img->cof[i][j][iii][jjj]=0; /* reset luma coeffs */
514
515 if(img->imod==INTRA_MB_NEW) /* read DC coeffs for new intra modes
*/
516 {
517     for (i=0;i<BLOCK_SIZE;i++)
518         for (j=0;j<BLOCK_SIZE;j++)
519             img->ipredmode[img->block_x+i+1][img->block_y+j+1]=0;
520
521     coef_ctr=-1;
522     len = 0; /* just to get inside the
loop */
523     for(k=0;(k<17) && (len!=1);k++)
524     {
525         get_symbol("DC luma 16x16",&len,&info,SE_LUM_DC_INTRA);
526
527         if(len>29) /* illegal length, start_scan sync search */
528         {
529             /*!
530              * \note
531              * Error concealment for illegal luminance coeffs:
532              * return EOB element to finalize block

```

```

534         */
535         set_ec_flag(SE_LUM_DC_INTRA);
536         get_concealed_element(&len, &info, SE_LUM_DC_INTRA);
537
538     }
539     if (len != 1)                /* leave if len=1 */
540     {
541         linfo_levrun_inter(len, info, &level, &run);
542         coef_ctr = coef_ctr + run + 1;
543
544         i0 = SNGL_SCAN[coef_ctr][0];
545         j0 = SNGL_SCAN[coef_ctr][1];
546
547         if (coef_ctr > 15)
548         {
549             /*!
550              * \note
551              *   Error concealment for luminance coeffs blocks
552              *   exceeding the maximum possible coeff number
553              *   within the block:
554              *   return EOB element to finalize block
555              */
556             set_ec_flag(SE_LUM_DC_INTRA);
557             get_concealed_element(&len, &info, SE_LUM_DC_INTRA);
558         }
559
560         img->cof[i0][j0][0][0] = level; /* add new intra DC
561         coeff */
562     }
563     itrans_2(img); /* transform new intra DC */
564 }
565
566 if (img->imod == INTRA_MB_OLD || img->imod == INTRA_MB_NEW)
567 {
568     for (i = 0; i < 4; i++)
569     {
570         ii = img->block_x + i;
571         i3 = ii / 2;
572         for (j = 0; j < 4; j++)
573         {
574             jj = img->block_y + j;
575             j3 = jj / 2;
576             loopb[ii+1][jj+1] = 3;
577             loopb[ii ][jj+1] = max(loopb[ii ][jj+1], 2);
578             loopb[ii+1][jj ] = max(loopb[ii+1][jj ], 2);
579             loopb[ii+2][jj+1] = max(loopb[ii+2][jj+1], 2);
580             loopb[ii+1][jj+2] = max(loopb[ii+1][jj+2], 2);
581
582             loopc[i3+1][j3+1] = 2;
583             loopc[i3 ][j3+1] = max(loopc[i3 ][j3+1], 1);
584             loopc[i3+1][j3 ] = max(loopc[i3+1][j3 ], 1);
585             loopc[i3+2][j3+1] = max(loopc[i3+2][j3+1], 1);
586             loopc[i3+1][j3+2] = max(loopc[i3+1][j3+2], 1);
587         }
588     }
589     else /* inter */
590     {
591         for (i = 0; i < 4; i++)
592         {
593             ii = img->block_x + i;
594             i3 = ii / 2;
595             for (j = 0; j < 4; j++)
596             {

```

```

597         jj=img->block_y+j;
598         j3=jj/2;
599
600         mvDiffX = img->mv[ii+4][jj][0] - img->mv[ii-1+4][jj][0]
601         ];
602         mvDiffY = img->mv[ii+4][jj][1] - img->mv[ii-1+4][jj][1]
603         ];
604
605         if ((mvDiffX*mvDiffX >= 16 || mvDiffY*mvDiffY >= 16)
606         && ii > 0)
607         {
608             loopb[ii ][jj+1]=max(loopb[ii ][jj+1],1);
609             loopb[ii+1][jj+1]=max(loopb[ii+1][jj+1],1);
610             loopc[i3 ][j3+1]=max(loopc[i3 ][j3+1],1);
611             loopc[i3+1][j3+1]=max(loopc[i3+1][j3+1],1);
612         }
613
614         mvDiffX = img->mv[ii+4][jj][0] - img->mv[ii+4][jj-1][0]
615         ];
616         mvDiffY = img->mv[ii+4][jj][1] - img->mv[ii+4][jj-1][1]
617         ];
618
619         if ((mvDiffX*mvDiffX >= 16 || mvDiffY*mvDiffY >= 16)
620         && jj > 0)
621         {
622             loopb[ii+1][jj ]=max(loopb[ii+1][jj ],1);
623             loopb[ii+1][jj+1]=max(loopb[ii+1][jj+1],1);
624             loopc[i3+1][j3 ]=max(loopc[i3+1][j3 ],1);
625             loopc[i3+1][j3+1]=max(loopc[i3+1][j3+1],1);
626         }
627     }
628 }
629
630 if (img->imod == INTRA_MB_OLD && img->qp < 24)
631     scan_mode=DOUBLE_SCAN;
632 else
633     scan_mode=SINGLE_SCAN;
634
635 /* luma coefficients */
636 for (block_y=0; block_y < 4; block_y += 2) /* all modes */
637 {
638     for (block_x=0; block_x < 4; block_x += 2)
639     {
640         for (j=block_y; j < block_y+2; j++)
641         {
642             jj=j/2;
643             for (i=block_x; i < block_x+2; i++)
644             {
645                 ii=i/2;
646                 if (img->imod == INTRA_MB_NEW)
647                     start_scan = 1; /* skip DC coeff */
648                 else
649                     start_scan = 0; /* take all coeffs */
650
651                 if((cbp & (int)pow(2,(ii+2*jj))) != 0) /* are
652                 there any coeff in current block at all */
653                 {
654                     if (scan_mode==SINGLE_SCAN)
655                     {
656                         coef_ctr=start_scan-1;
657                         len = 0; /*
658                         just to get inside the loop */
659                         for(k=start_scan;(k<17) && (len!=1);k++)
660                         {

```

```

655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707

```

```

/*
 * make distinction between INTRA and
 * INTER coded
 * luminance coefficients
 */
if (interintra == INTER_CODED_MB )
{
    if (k == 0)
        se = SE_LUM_DC_INTER;
    else
        se = SE_LUM_AC_INTER;
}
else
{
    if (k == 0)
        se = SE_LUM_DC_INTRA;
    else
        se = SE_LUM_AC_INTRA;
}
get_symbol("Luma sng",&len,&info,se);

if(len>29) /* illegal length, do
error concealment */
{
    /*!
     * \note
     * Error concealment for illegal
     * luminance coeffs:
     * return EOB element to
     * finalize block
     */
    set_ec_flag(se);
    get_concealed_element(&len, &info,
        se);
}
if (len != 1) /* leave if len=1
*/
{
    linfo_levrun_inter(len,info,&level
        ,&run);
    coef_ctr += run+1;

    i0=SNGL_SCAN[coef_ctr][0];
    j0=SNGL_SCAN[coef_ctr][1];

    if(coef_ctr > 15)
    {
        /*!
         * \note
         * Error concealment for
         * luminance coeffs blocks
         * exceeding the maximum
         * possible coeff number within
         * the block:
         * return EOB element to
         * finalize block
         */
        set_ec_flag(se);
        get_concealed_element(&len, &
            info, se);
    }

    img->cof[i][j][i0][j0]=level*JQ1[
    img->qp];
    if (level!=0)
    {

```

```

708         loopb[img->block_x+i+1][img->
        block_y+j+1]=max(loopb[img->
        block_x+i+1][img->block_y+j+1
        ],2);
709         loopb[img->block_x+i ][img->
        block_y+j+1]=max(loopb[img->
        block_x+i ][img->block_y+j+1
        ],1);
710         loopb[img->block_x+i+1][img->
        block_y+j ]=max(loopb[img->
        block_x+i+1][img->block_y+j
        ],1);
711         loopb[img->block_x+i+2][img->
        block_y+j+1]=max(loopb[img->
        block_x+i+2][img->block_y+j+1
        ],1);
712         loopb[img->block_x+i+1][img->
        block_y+j+2]=max(loopb[img->
        block_x+i+1][img->block_y+j+2
        ],1);
713     }
714 }
715 }
716 }
717 else /* double scan (old intra with QP<24*/
718 {
719     for(scan_loop_ctr=0;scan_loop_ctr<2;
        scan_loop_ctr++)
720     {
721         coef_ctr=start_scan-1;
722         len=0; /*
723         just to get inside the loop */
724         for(k=0; k<9 && len!=1;k++)
725         {
726             /* <pu> distinction between DC
727             and AC luminance coeffs */
728             if ( k == 0 )
729                 se = SE_LUM_DC_INTRA;
730             else
731                 se = SE_LUM_AC_INTRA;
732             get_symbol("Luma dbl",&len,&info,
733             se);
734
735             if(len>29) /* illegal
736             length, do error concealment */
737             {
738                 /*!
739                 * \note
740                 * Error concealment for
741                 illegal luminance coeffs:
742                 * return EOB element to
743                 finalize block
744                 */
745                 set_ec_flag(se);
746                 get_concealed_element(&len, &
747                 info, se);
748             }
749             if (len != 1) /* leave if
750             len=1 */
751             {
752                 linfo_levrun_intra(len,info,&
753                 level,&run);
754                 coef_ctr=coef_ctr+run+1;
755                 i0=DBL_SCAN[coef_ctr][0][

```

```

749     scan_loop_ctr];
750     j0=DBL_SCAN[coef_ctr][1][
751     scan_loop_ctr];

752     if(coef_ctr > 7) /*
753     illegal number of coeffs, do
754     error concealment */
755     {
756         /*!
757         * \note
758         * Error concealment for
759         luminance coeffs blocks
760         * exceeding the maximum
761         possible coeff number
762         within the block:
763         * return EOB element to
764         finalize block
765         */
766         set_ec_flag(se);
767         get_concealed_element(&len
768         , &info, se);
769     }
770     img->cof[i][j][i0][j0]=level*
771     JQ1[img->qp];

772     if (level!=0)
773     {
774         loopb[img->block_x+i+1][
775         img->block_y+j+1]=max(
776         loopb[img->block_x+i+1][
777         img->block_y+j+1],2);
778         loopb[img->block_x+i ][
779         img->block_y+j+1]=max(
780         loopb[img->block_x+i ][
781         img->block_y+j+1],1);
782         loopb[img->block_x+i+1][
783         img->block_y+j ]=max(
784         loopb[img->block_x+i+1][
785         img->block_y+j ],1);
786         loopb[img->block_x+i+2][
787         img->block_y+j+1]=max(
788         loopb[img->block_x+i+2][
789         img->block_y+j+1],1);
790         loopb[img->block_x+i+1][
791         img->block_y+j+2]=max(
792         loopb[img->block_x+i+1][
793         img->block_y+j+2],1);
794     }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

```

790
791     /* chroma 2x2 DC coeff */
792     if(cbp>15)
793     {
794         for (ll=0;ll<3;ll+=2)
795         {
796             for (i=0;i<4;i++)
797                 img->cofu[i]=0;
798
799             coef_ctr=-1;
800             len=0;
801             for(k=0;(k<5)&&(len!=1);k++)
802             {
803                 /* <pu> distinct between INTRA and INTER coded
804                  elements */
805                 if ( interintra == INTRA_CODED_MB)
806                     se = SE_CHR_DC_INTRA;
807                 else
808                     se = SE_CHR_DC_INTER;
809                 get_symbol("2x2 DC Chroma",&len,&info,se);
810
811                 if (len>29) /* illegal length do error
812                  concealment */
813                 {
814                     /*!
815                      * \note
816                      * Error concealment for illegal chrominance
817                      coeffs:
818                      * return EOB element to finalize block
819                      */
820                     set_ec_flag(se);
821                     get_concealed_element(&len, &info, se);
822                 }
823                 if (len != 1)
824                 {
825                     linfo_levrun_c2x2(len,info,&level,&run);
826                     coef_ctr=coef_ctr+run+1;
827                     if(coef_ctr>3) /* illegal number of coeffs,
828                      do error concealment */
829                     {
830                         /*!
831                          * \note
832                          * Error concealment for chrominance coeffs
833                          blocks
834                          * exceeding the maximum possible coeff
835                          number within the block:
836                          * return EOB element to finalize block
837                          */
838                         set_ec_flag(se);
839                         get_concealed_element(&len, &info, se);
840                     }
841                     img->cofu[coef_ctr]=level*JQ1[QP_SCALE_CR[img->qp
842                     ]];
843
844                     if (level != 0 );
845                     {
846                         for (j=0;j<2;j++)
847                         {
848                             for (i=0;i<2;i++)
849                             {
850                                 loopc[m2+i+1][jg2+j+1]=max(loopc[m2+i+
851                                 1][jg2+j+1],2);
852                             }
853                         }
854
855                         for (i=0;i<2;i++)

```

```

848         {
849             loopc[m2+i+1][jg2      ]=max(loopc[m2+i+1][
850             jg2      ],1);
851             loopc[m2+i+1][jg2+3    ]=max(loopc[m2+i+1][
852             jg2+3    ],1);
853             loopc[m2      ][jg2+i+1]=max(loopc[m2      ][
854             jg2+i+1],1);
855             loopc[m2+3    ][jg2+i+1]=max(loopc[m2+3    ][
856             jg2+i+1],1);
857         }
858     }
859 }
860
861 img->cof[0+11][4][0][0]=(img->cofu[0]+img->cofu[1]+img->
862 cofu[2]+img->cofu[3])/2;
863
864 img->cof[1+11][4][0][0]=(img->cofu[0]-img->cofu[1]+img->
865 cofu[2]-img->cofu[3])/2;
866
867 img->cof[0+11][5][0][0]=(img->cofu[0]+img->cofu[1]-img->
868 cofu[2]-img->cofu[3])/2;
869
870 img->cof[1+11][5][0][0]=(img->cofu[0]-img->cofu[1]-img->
871 cofu[2]+img->cofu[3])/2;
872
873 }
874
875 /* chroma AC coeff, all zero fram start_scan */
876 if (cbp>31)
877 {
878     block_y=4;
879     for (block_x=0; block_x < 4; block_x += 2)
880     {
881         for (j=block_y; j < block_y+2; j++)
882         {
883             jj=j/2;
884             j1=j-4;
885             for (i=block_x; i < block_x+2; i++)
886             {
887                 ii=i/2;
888                 i1=i%2;
889                 {
890                     coef_ctr=0;
891                     len=0;
892                     for(k=0; (k<16)&&(len!=1);k++)
893                     {
894                         /* <pu> distinct between Intra and Inter
895                         coded elements */
896                         if ( interintra == INTER_CODED_MB )
897                             se = SE_CHR_AC_INTER;
898                         else
899                             se = SE_CHR_AC_INTRA;
900                         get_symbol("AC Chroma",&len,&info,se);
901
902                         if (len>29) /* illegal length, do error
903                         concealment */
904                         {
905                             /*!
906                             * \note
907                             * Error concealment for illegal
908                             chrominance coeffs:
909                             * return EOB element to finalize
910                             block
911                             */
912                             set_ec_flag(se);
913                             get_concealed_element(&len, &info, se);
914                         }
915                     }
916                     if (len != 1)
917                     {

```

```

902         linfo_levrun_inter(len,info,&level,&
          run);
903         coef_ctr=coef_ctr+run+1;
904         i0=SNGL_SCAN[coef_ctr][0];
905         j0=SNGL_SCAN[coef_ctr][1];
906         if(coef_ctr>15) /* illegal number of
          coeffs, do error concealment */
          {
907             /*!
908             * \note
909             * Error concealment for
910             * chrominance coeffs blocks
911             * exceeding the maximum
912             * possible coeff number within the
913             * block:
914             * return EOB element to
915             * finalize block
916             */
917             set_ec_flag(se);
918             get_concealed_element(&len, &info,
919                                   se);
920         }
921         img->cof[i][j][i0][j0]=level*JQ1[
          QP_SCALE_CR[img->qp]];
922
923         if (level!=0)
924         {
925             loopc[m2+i1+1][jg2+j1+1]=max(loopc
          [m2+i1+1][jg2+j1+1],2);
926
927             loopc[m2+i1 ][jg2+j1+1]=max(loopc
          [m2+i1 ][jg2+j1+1],1);
928             loopc[m2+i1+1][jg2+j1 ]=max(loopc
          [m2+i1+1][jg2+j1 ],1);
929             loopc[m2+i1+2][jg2+j1+1]=max(loopc
          [m2+i1+2][jg2+j1+1],1);
930             loopc[m2+i1+1][jg2+j1+2]=max(loopc
          [m2+i1+1][jg2+j1+2],1);
931         }
932     }
933 }
934
935 /* start_scan decoding */
936
937 /* luma */
938 if (img->imod==INTRA_MB_NEW)
939     intrapred_luma_2(img,kmod);
940
941 for(j=0;j<MB_BLOCK_SIZE/BLOCK_SIZE;j++)
942 {
943     joff=j*4;
944     j4=img->block_y+j;
945     for(i=0;i<MB_BLOCK_SIZE/BLOCK_SIZE;i++)
946     {
947         ioff=i*4;
948         i4=img->block_x+i;
949         if(img->imod==INTRA_MB_OLD)
950         {
951             if (intrapred(img,ioff,joff,i4,j4)==SEARCH_SYNC) /*
          make 4x4 prediction block mpr from given prediction
          img->mb_mode */

```

```

953         return SEARCH_SYNC;                                /* bit
954         error */
955     }
956     else if(img->imod==INTRA_MB_INTER)
957     {
958         for(ii=0;ii<BLOCK_SIZE;ii++)
959         {
960             vec2_x=(i4*4+ii)*4;
961             vec1_x=vec2_x+img->mv[i4+BLOCK_SIZE][j4][0];
962             for(jj=0;jj<MB_BLOCK_SIZE/BLOCK_SIZE;jj++)
963             {
964                 vec2_y=(j4*4+jj)*4;
965                 vec1_y=vec2_y+img->mv[i4+4][j4][1];
966                 img->mpr[ii+ioff][jj+joff]=get_pixel(ref_frame
967                 ,vec1_x,vec1_y,img);
968                 //
969                 img->mpr[ii+ioff][jj+joff]=mref[ref_frame][vec1
970                 _y][vec1_x];
971             }
972         }
973     }
974     itrans(img,ioff,joff,i,j);    /* use DCT transform and
975     make 4x4 block m7 from prediction block mpr */
976
977     for(ii=0;ii<BLOCK_SIZE;ii++)
978     {
979         for(jj=0;jj<BLOCK_SIZE;jj++)
980         {
981             imgY[j4*BLOCK_SIZE+jj][i4*BLOCK_SIZE+ii]=img->m7[
982             ii][jj]; /* construct picture from 4x4 blocks*/
983         }
984     }
985 }
986 /* chroma */
987
988 for(uv=0;uv<2;uv++)
989 {
990     if (img->imod==INTRA_MB_OLD || img->imod==INTRA_MB_NEW) /*
991     intra mode */
992     {
993         js0=0;
994         js1=0;
995         js2=0;
996         js3=0;
997         for(i=0;i<4;i++)
998         {
999             if(mb_available_up)
1000             {
1001                 js0=js0+imgUV[uv][img->pix_c_y-1][img->pix_c_x+i];
1002                 js1=js1+imgUV[uv][img->pix_c_y-1][img->pix_c_x+i+4
1003                 ];
1004             }
1005             if(mb_available_left)
1006             {
1007                 js2=js2+imgUV[uv][img->pix_c_y+i][img->pix_c_x-1];
1008                 js3=js3+imgUV[uv][img->pix_c_y+i+4][img->pix_c_x-1
1009                 ];
1010             }
1011         }
1012     }
1013     if(mb_available_up && mb_available_left)
1014     {
1015         js[0][0]=(js0+js2+4)/8;
1016         js[1][0]=(js1+2)/4;
1017         js[0][1]=(js3+2)/4;

```

```

1010         js[1][1]=(js1+js3+4)/8;
1011     }
1012     if(mb_available_up && !mb_available_left)
1013     {
1014         js[0][0]=(js0+2)/4;
1015         js[1][0]=(js1+2)/4;
1016         js[0][1]=(js0+2)/4;
1017         js[1][1]=(js1+2)/4;
1018     }
1019     if(mb_available_left && !mb_available_up)
1020     {
1021         js[0][0]=(js2+2)/4;
1022         js[1][0]=(js2+2)/4;
1023         js[0][1]=(js3+2)/4;
1024         js[1][1]=(js3+2)/4;
1025     }
1026     if(!mb_available_up && !mb_available_left)
1027     {
1028         js[0][0]=128;
1029         js[1][0]=128;
1030         js[0][1]=128;
1031         js[1][1]=128;
1032     }
1033 }
1034
1035 for (j=4;j<6;j++)
1036 {
1037     joff=(j-4)*4;
1038     j4=img->pix_c_y+joff;
1039     for (i=0;i<2;i++)
1040     {
1041         ioff=i*4;
1042         i4=img->pix_c_x+ioff;
1043         /* make pred */
1044         if(img->imod==INTRA_MB_OLD || img->imod==INTRA_MB_NEW)
1045             /* intra */
1046         {
1047             for (ii=0;ii<4;ii++)
1048                 for (jj=0;jj<4;jj++)
1049                 {
1050                     img->mpr[ii+ioff][jj+joff]=js[i][j-4];
1051                 }
1052         }
1053     else
1054     {
1055         for (jj=0;jj<4;jj++)
1056         {
1057             jf=(j4+jj)/2;
1058             for (ii=0;ii<4;ii++)
1059             {
1060                 if1=(i4+ii)/2;
1061                 i1=(img->pix_c_x+ii+ioff)*8+img->mv[if1+4][jf][0];
1062                 j1=(img->pix_c_y+jj+joff)*8+img->mv[if1+4][jf][1];
1063                 ii0=i1/8;
1064                 jj0=j1/8;
1065                 ii1=(i1+7)/8;
1066                 jj1=(j1+7)/8;
1067                 if1=(i1 & 7);
1068                 jf1=(j1 & 7);
1069                 if0=8-if1;
1070                 jf0=8-jf1;
1071                 img->mpr[ii+ioff][jj+joff]=(if0*jf0*mcef[

```

```

ref_frame][uv][jj0][ii0]+
1073 if1*jf0*mcef[
ref_frame][uv
1074 ][jj0][ii1]+
if0*jf1*mcef[
ref_frame][uv
1075 ][jj1][ii0]+
if1*jf1*mcef[
ref_frame][uv
][jj1][ii1]+32
)/64;

1076 }
1077 }
1078 }
1079 itrans(img,ioff,joff,2*uv+i,j);
1080 for(ii=0;ii<4;ii++)
1081     for(jj=0;jj<4;jj++)
1082     {
1083         imgUV[uv][j4+jj][i4+ii]=img->m7[ii][jj];
1084     }
1085 }
1086 }
1087 }
1088 return DECODING_OK;
1089 }
1090
1091
1092 /*!
1093 * \fn      inter_intra()
1094 * \brief   returns coding mode of actual macroblock if it is either
1095 *          inter or intra coded
1096 * \return  INTER_CODED_MB, macroblock is inter coded
1097 *          INTRA_CODED_MB, macroblock is intra coded,
1098 *          either with the old or new intra mode
1099 * <pu> added 6.10.2000
1100 */
1101
1102 int inter_intra(
1103     struct img_par *img) /*!< pointer to image struct */
1104 {
1105     if ( img->imod == INTRA_MB_OLD ||  img->imod == INTRA_MB_NEW ) //
1106         bug fix, DM 14/03/01
1107     {
1108         return INTRA_CODED_MB;
1109     }
1110     else
1111     {
1112         if ( img->mb_mode < 8)
1113             return INTER_CODED_MB;
1114         else
1115             return INTRA_CODED_MB;
1116     }
1117 }
1118

```