CODE
by
R.Mutt

dcraw.c

```
/*
   dcraw.c -- Dave Coffin's raw photo decoder
   Copyright 1997-2018 by Dave Coffin, dcoffin a cybercom o net

   This is a command-line ANSI C program to convert raw photos from
   any digital camera on any computer running any operating system.

   No license is required to download and use dcraw.c.  However,
   to lawfully redistribute dcraw, you must either (a) offer, at
   no extra charge, full source code* for all executable files
   containing RESTRICTED functions, (b) distribute this code under
   the GPL Version 2 or later, (c) remove all RESTRICTED functions,
   re-implement them, or copy them from an earlier, unrestricted
   Revision of dcraw.c, or (d) purchase a license from the author.

   The functions that process Foveon images have been RESTRICTED
   since Revision 1.237.  All other code remains free for all uses.

   *If you have not modified dcraw.c in any way, a link to my
   homepage qualifies as "full source code".

   $Revision: 1.478 $
   $Date: 2018/06/01 20:36:25 $
 */

#define DCRAW_VERSION "9.28"

#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#define _USE_MATH_DEFINES
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <float.h>
#include <limits.h>
#include <math.h>
#include <setjmp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>

#if defined(DJGPP) || defined(__MINGW32__)
#define fseeko fseek
#define ftello ftell
#else
#define fgetc getc_unlocked
#endif
#ifdef __CYGWIN__
#include <io.h>
#endif
#ifdef WIN32
#include <sys/utime.h>
#include <winsock2.h>
#pragma comment(lib, "ws2_32.lib")
#define snprintf _snprintf
#define strcasecmp stricmp
#define strncasecmp strnicmp
```

```
61.  typedef __int64 INT64;
62.  typedef unsigned __int64 UINT64;
63.  #else
64.  #include <unistd.h>
65.  #include <utime.h>
66.  #include <netinet/in.h>
67.  typedef long long INT64;
68.  typedef unsigned long long UINT64;
69.  #endif
70.
71.  #ifdef NODEPS
72.  #define NO_JASPER
73.  #define NO_JPEG
74.  #define NO_LCMS
75.  #endif
76.  #ifndef NO_JASPER
77.  #include <jasper/jasper.h>      /* Decode Red camera movies */
78.  #endif
79.  #ifndef NO_JPEG
80.  #include <jpeglib.h>            /* Decode compressed Kodak DC120 photos */
81.  #endif                          /* and Adobe Lossy DNGs */
82.  #ifndef NO_LCMS
83.  #include <lcms2.h>              /* Support color profiles */
84.  #endif
85.  #ifdef LOCALEDIR
86.  #include <libintl.h>
87.  #define _(String) gettext(String)
88.  #else
89.  #define _(String) (String)
90.  #endif
91.
92.  #if !defined(uchar)
93.  #define uchar unsigned char
94.  #endif
95.  #if !defined(ushort)
96.  #define ushort unsigned short
97.  #endif
98.
99.  /*
100.    All global variables are defined here, and all functions that
101.    access them are prefixed with "CLASS".  For thread-safety, all
102.    non-const static local variables except cbrt[] must be declared
103.    "thread_local".
104.  */
105. FILE *ifp, *ofp;
106. short order;
107. const char *ifname;
108. char *meta_data, xtrans[6][6], xtrans_abs[6][6];
109. char cdesc[5], desc[512], make[64], model[64], model2[64], artist[64];
110. float flash_used, canon_ev, iso_speed, shutter, aperture, focal_len;
111. time_t timestamp;
112. off_t strip_offset, data_offset;
113. off_t thumb_offset, meta_offset, profile_offset;
114. unsigned shot_order, kodak_cbpp, exif_cfa, unique_id;
115. unsigned thumb_length, meta_length, profile_length;
116. unsigned thumb_misc, *oprof, fuji_layout, shot_select=0, multi_out=0;
117. unsigned tiff_nifds, tiff_samples, tiff_bps, tiff_compress;
118. unsigned black, maximum, mix_green, raw_color, zero_is_bad;
119. unsigned zero_after_ff, is_raw, dng_version, is_foveon, data_error;
120. unsigned tile_width, tile_length, gpsdata[32], load_flags;
121. unsigned flip, tiff_flip, filters, colors;
122. ushort raw_height, raw_width, height, width, top_margin, left_margin;
123. ushort shrink, iheight, iwidth, fuji_width, thumb_width, thumb_height;
124. ushort *raw_image, (*image)[4], cblack[4102];
125. ushort white[8][8], curve[0x10000], cr2_slice[3], sraw_mul[4];
```

```
126. double pixel_aspect, aber[4]={1,1,1,1}, gamm[6]={ 0.45,4.5,0,0,0,0 };
127. float bright=1, user_mul[4]={0,0,0,0}, threshold=0;
128. int mask[8][4];
129. int half_size=0, four_color_rgb=0, document_mode=0, highlight=0;
130. int verbose=0, use_auto_wb=0, use_camera_wb=0, use_camera_matrix=1;
131. int output_color=1, output_bps=8, output_tiff=0, med_passes=0;
132. int no_auto_bright=0;
133. unsigned greybox[4] = { 0, 0, UINT_MAX, UINT_MAX };
134. float cam_mul[4], pre_mul[4], cmatrix[3][4], rgb_cam[3][4];
135. const double xyz_rgb[3][3] = {                /* XYZ from RGB */
136.   { 0.412453, 0.357580, 0.180423 },
137.   { 0.212671, 0.715160, 0.072169 },
138.   { 0.019334, 0.119193, 0.950227 } };
139. const float d65_white[3] = { 0.950456, 1, 1.088754 };
140. int histogram[4][0x2000];
141. void (*write_thumb)(), (*write_fun)();
142. void (*load_raw)(), (*thumb_load_raw)();
143. jmp_buf failure;
144.
145. struct decode {
146.   struct decode *branch[2];
147.   int leaf;
148. } first_decode[2048], *second_decode, *free_decode;
149.
150. struct tiff_ifd {
151.   int width, height, bps, comp, phint, offset, flip, samples, bytes;
152.   int tile_width, tile_length;
153.   float shutter;
154. } tiff_ifd[10];
155.
156. struct ph1 {
157.   int format, key_off, tag_21a;
158.   int black, split_col, black_col, split_row, black_row;
159.   float tag_210;
160. } ph1;
161.
162. #define CLASS
163.
164. #define FORC(cnt) for (c=0; c < cnt; c++)
165. #define FORC3 FORC(3)
166. #define FORC4 FORC(4)
167. #define FORCC FORC(colors)
168.
169. #define SQR(x) ((x)*(x))
170. #define ABS(x) (((int)(x) ^ ((int)(x) >> 31)) - ((int)(x) >> 31))
171. #define MIN(a,b) ((a) < (b) ? (a) : (b))
172. #define MAX(a,b) ((a) > (b) ? (a) : (b))
173. #define LIM(x,min,max) MAX(min,MIN(x,max))
174. #define ULIM(x,y,z) ((y) < (z) ? LIM(x,y,z) : LIM(x,z,y))
175. #define CLIP(x) LIM((int)(x),0,65535)
176. #define SWAP(a,b) { a=a+b; b=a-b; a=a-b; }
177.
178. /*
179.    In order to inline this calculation, I make the risky
180.    assumption that all filter patterns can be described
181.    by a repeating pattern of eight rows and two columns
182.
183.    Do not use the FC or BAYER macros with the Leaf CatchLight,
184.    because its pattern is 16x16, not 2x8.
185.
186.    Return values are either 0/1/2/3 = G/M/C/Y or 0/1/2/3 = R/G1/B/G2
187.
188.         PowerShot 600   PowerShot A50   PowerShot Pro70 Pro90 & G1
189.         0xe1e4e1e4:     0x1b4e4b1e:     0x1e4b4e1b:     0xb4b4b4b4:
190.
```

```
191.           0 1 2 3 4 5     0 1 2 3 4 5     0 1 2 3 4 5     0 1 2 3 4 5
192.         0 G M G M G M   0 C Y C Y C Y   0 Y C Y C Y C   0 G M G M G M
193.         1 C Y C Y C Y   1 M G M G M G   1 M G M G M G   1 Y C Y C Y C
194.         2 M G M G M G   2 Y C Y C Y C   2 C Y C Y C Y
195.         3 C Y C Y C Y   3 G M G M G M   3 G M G M G M
196.                         4 C Y C Y C Y   4 Y C Y C Y C
197.         PowerShot A5    5 G M G M G M   5 G M G M G M
198.         0x1e4e1e4e:     6 Y C Y C Y C   6 C Y C Y C Y
199.                         7 M G M G M G   7 M G M G M G
200.           0 1 2 3 4 5
201.         0 C Y C Y C Y
202.         1 G M G M G M
203.         2 C Y C Y C Y
204.         3 M G M G M G
205.
206.    All RGB cameras use one of these Bayer grids:
207.
208.         0x16161616:     0x61616161:     0x49494949:     0x94949494:
209.
210.           0 1 2 3 4 5     0 1 2 3 4 5     0 1 2 3 4 5     0 1 2 3 4 5
211.         0 B G B G B G   0 G R G R G R   0 G B G B G B   0 R G R G R G
212.         1 G R G R G R   1 B G B G B G   1 R G R G R G   1 G B G B G B
213.         2 B G B G B G   2 G R G R G R   2 G B G B G B   2 R G R G R G
214.         3 G R G R G R   3 B G B G B G   3 R G R G R G   3 G B G B G B
215.  */
216.
217. #define RAW(row,col) \
218.         raw_image[(row)*raw_width+(col)]
219.
220. #define FC(row,col) \
221.         (filters >> ((((row) << 1 & 14) + ((col) & 1)) << 1) & 3)
222.
223. #define BAYER(row,col) \
224.         image[((row) >> shrink)*iwidth + ((col) >> shrink)][FC(row,col)]
225.
226. #define BAYER2(row,col) \
227.         image[((row) >> shrink)*iwidth + ((col) >> shrink)][fcol(row,col)]
228.
229. int CLASS fcol (int row, int col)
230. {
231.   static const char filter[16][16] =
232.   { { 2,1,1,3,2,3,2,0,3,2,3,0,1,2,1,0 },
233.     { 0,3,0,2,0,1,3,1,0,1,1,2,0,3,3,2 },
234.     { 2,3,3,2,3,1,1,3,3,1,2,1,2,0,0,3 },
235.     { 0,1,0,1,0,2,0,2,2,0,3,0,1,3,2,1 },
236.     { 3,1,1,2,0,1,0,2,1,3,1,3,0,1,3,0 },
237.     { 2,0,0,3,3,2,3,1,2,0,2,0,3,2,2,1 },
238.     { 2,3,3,1,2,1,2,1,2,1,1,2,3,0,0,1 },
239.     { 1,0,0,2,3,0,0,3,0,3,0,3,2,1,2,3 },
240.     { 2,3,3,1,1,2,1,0,3,2,3,0,2,3,1,3 },
241.     { 1,0,2,0,3,0,3,2,0,1,1,2,0,1,0,2 },
242.     { 0,1,1,3,3,2,2,1,1,3,3,0,2,1,3,2 },
243.     { 2,3,2,0,0,1,3,0,2,0,1,2,3,0,1,0 },
244.     { 1,3,1,2,3,2,3,2,0,2,0,1,1,0,3,0 },
245.     { 0,2,0,3,1,0,0,1,1,3,3,2,3,2,2,1 },
246.     { 2,1,3,2,3,1,2,1,0,3,0,2,0,2,0,2 },
247.     { 0,3,1,0,0,2,0,3,2,1,3,1,1,3,1,3 } };
248.
249.   if (filters == 1) return filter[(row+top_margin)&15][(col+left_margin)&15];
250.   if (filters == 9) return xtrans[(row+6) % 6][(col+6) % 6];
251.   return FC(row,col);
252. }
253.
254. #ifndef __GLIBC__
255. char *my_memmem (char *haystack, size_t haystacklen,
```

```
256.                char *needle, size_t needlelen)
257. {
258.   char *c;
259.   for (c = haystack; c <= haystack + haystacklen - needlelen; c++)
260.     if (!memcmp (c, needle, needlelen))
261.       return c;
262.   return 0;
263. }
264. #define memmem my_memmem
265. char *my_strcasestr (char *haystack, const char *needle)
266. {
267.   char *c;
268.   for (c = haystack; *c; c++)
269.     if (!strncasecmp(c, needle, strlen(needle)))
270.       return c;
271.   return 0;
272. }
273. #define strcasestr my_strcasestr
274. #endif
275.
276. void CLASS merror (void *ptr, const char *where)
277. {
278.   if (ptr) return;
279.   fprintf (stderr,_("%s: Out of memory in %s\n"), ifname, where);
280.   longjmp (failure, 1);
281. }
282.
283. void CLASS derror()
284. {
285.   if (!data_error) {
286.     fprintf (stderr, "%s: ", ifname);
287.     if (feof(ifp))
288.       fprintf (stderr,_("Unexpected end of file\n"));
289.     else
290.       fprintf (stderr,_("Corrupt data near 0x%llx\n"), (INT64) ftello(ifp));
291.   }
292.   data_error++;
293. }
294.
295. ushort CLASS sget2 (uchar *s)
296. {
297.   if (order == 0x4949)          /* "II" means little-endian */
298.     return s[0] | s[1] << 8;
299.   else                          /* "MM" means big-endian */
300.     return s[0] << 8 | s[1];
301. }
302.
303. ushort CLASS get2()
304. {
305.   uchar str[2] = { 0xff,0xff };
306.   fread (str, 1, 2, ifp);
307.   return sget2(str);
308. }
309.
310. unsigned CLASS sget4 (uchar *s)
311. {
312.   if (order == 0x4949)
313.     return s[0] | s[1] << 8 | s[2] << 16 | s[3] << 24;
314.   else
315.     return s[0] << 24 | s[1] << 16 | s[2] << 8 | s[3];
316. }
317. #define sget4(s) sget4((uchar *)s)
318.
319. unsigned CLASS get4()
320. {
```

```
  uchar str[4] = { 0xff,0xff,0xff,0xff };
  fread (str, 1, 4, ifp);
  return sget4(str);
}

unsigned CLASS getint (int type)
{
  return type == 3 ? get2() : get4();
}

float CLASS int_to_float (int i)
{
  union { int i; float f; } u;
  u.i = i;
  return u.f;
}

double CLASS getreal (int type)
{
  union { char c[8]; double d; } u;
  int i, rev;

  switch (type) {
    case 3: return (unsigned short) get2();
    case 4: return (unsigned int) get4();
    case 5:  u.d = (unsigned int) get4();
      return u.d / (unsigned int) get4();
    case 8: return (signed short) get2();
    case 9: return (signed int) get4();
    case 10: u.d = (signed int) get4();
      return u.d / (signed int) get4();
    case 11: return int_to_float (get4());
    case 12:
      rev = 7 * ((order == 0x4949) == (ntohs(0x1234) == 0x1234));
      for (i=0; i < 8; i++)
        u.c[i ^ rev] = fgetc(ifp);
      return u.d;
    default: return fgetc(ifp);
  }
}

void CLASS read_shorts (ushort *pixel, int count)
{
  if (fread (pixel, 2, count, ifp) < count) derror();
  if ((order == 0x4949) == (ntohs(0x1234) == 0x1234))
    swab (pixel, pixel, count*2);
}

void CLASS cubic_spline (const int *x_, const int *y_, const int len)
{
  float **A, *b, *c, *d, *x, *y;
  int i, j;

  A = (float **) calloc (((2*len + 4)*sizeof **A + sizeof *A), 2*len);
  if (!A) return;
  A[0] = (float *) (A + 2*len);
  for (i = 1; i < 2*len; i++)
    A[i] = A[0] + 2*len*i;
  y = len + (x = i + (d = i + (c = i + (b = A[0] + i*i))));
  for (i = 0; i < len; i++) {
    x[i] = x_[i] / 65535.0;
    y[i] = y_[i] / 65535.0;
  }
  for (i = len-1; i > 0; i--) {
    b[i] = (y[i] - y[i-1]) / (x[i] - x[i-1]);
```

```
    d[i-1] = x[i] - x[i-1];
  }
  for (i = 1; i < len-1; i++) {
    A[i][i] = 2 * (d[i-1] + d[i]);
    if (i > 1) {
      A[i][i-1] = d[i-1];
      A[i-1][i] = d[i-1];
    }
    A[i][len-1] = 6 * (b[i+1] - b[i]);
  }
  for(i = 1; i < len-2; i++) {
    float v = A[i+1][i] / A[i][i];
    for(j = 1; j <= len-1; j++)
      A[i+1][j] -= v * A[i][j];
  }
  for(i = len-2; i > 0; i--) {
    float acc = 0;
    for(j = i; j <= len-2; j++)
      acc += A[i][j]*c[j];
    c[i] = (A[i][len-1] - acc) / A[i][i];
  }
  for (i = 0; i < 0x10000; i++) {
    float x_out = (float)(i / 65535.0);
    float y_out = 0;
    for (j = 0; j < len-1; j++) {
      if (x[j] <= x_out && x_out <= x[j+1]) {
        float v = x_out - x[j];
        y_out = y[j] +
          ((y[j+1] - y[j]) / d[j] - (2 * d[j] * c[j] + c[j+1] * d[j])/6) * v
           + (c[j] * 0.5) * v*v + ((c[j+1] - c[j]) / (6 * d[j])) * v*v*v;
      }
    }
    curve[i] = y_out < 0.0 ? 0 : (y_out >= 1.0 ? 65535 :
                  (ushort)(y_out * 65535.0 + 0.5));
  }
  free (A);
}

void CLASS canon_600_fixed_wb (int temp)
{
  static const short mul[4][5] = {
    {  667, 358,397,565,452 },
    {  731, 390,367,499,517 },
    { 1119, 396,348,448,537 },
    { 1399, 485,431,508,688 } };
  int lo, hi, i;
  float frac=0;

  for (lo=4; --lo; )
    if (*mul[lo] <= temp) break;
  for (hi=0; hi < 3; hi++)
    if (*mul[hi] >= temp) break;
  if (lo != hi)
    frac = (float) (temp - *mul[lo]) / (*mul[hi] - *mul[lo]);
  for (i=1; i < 5; i++)
    pre_mul[i-1] = 1 / (frac * mul[hi][i] + (1-frac) * mul[lo][i]);
}

/* Return values:  0 = white  1 = near white  2 = not white */
int CLASS canon_600_color (int ratio[2], int mar)
{
  int clipped=0, target, miss;

  if (flash_used) {
    if (ratio[1] < -104)
```

```
      { ratio[1] = -104; clipped = 1; }
    if (ratio[1] >   12)
      { ratio[1] =   12; clipped = 1; }
  } else {
    if (ratio[1] < -264 || ratio[1] > 461) return 2;
    if (ratio[1] < -50)
      { ratio[1] = -50; clipped = 1; }
    if (ratio[1] > 307)
      { ratio[1] = 307; clipped = 1; }
  }
  target = flash_used || ratio[1] < 197
         ? -38 - (398 * ratio[1] >> 10)
         : -123 + (48 * ratio[1] >> 10);
  if (target - mar <= ratio[0] &&
      target + 20  >= ratio[0] && !clipped) return 0;
  miss = target - ratio[0];
  if (abs(miss) >= mar*4) return 2;
  if (miss < -20) miss = -20;
  if (miss > mar) miss = mar;
  ratio[0] = target - miss;
  return 1;
}

void CLASS canon_600_auto_wb()
{
  int mar, row, col, i, j, st, count[] = { 0,0 };
  int test[8], total[2][8], ratio[2][2], stat[2];

  memset (&total, 0, sizeof total);
  i = canon_ev + 0.5;
  if      (i < 10) mar = 150;
  else if (i > 12) mar = 20;
  else mar = 280 - 20 * i;
  if (flash_used) mar = 80;
  for (row=14; row < height-14; row+=4)
    for (col=10; col < width; col+=2) {
      for (i=0; i < 8; i++)
        test[(i & 4) + FC(row+(i >> 1),col+(i & 1))] =
                     BAYER(row+(i >> 1),col+(i & 1));
      for (i=0; i < 8; i++)
        if (test[i] < 150 || test[i] > 1500) goto next;
      for (i=0; i < 4; i++)
        if (abs(test[i] - test[i+4]) > 50) goto next;
      for (i=0; i < 2; i++) {
        for (j=0; j < 4; j+=2)
          ratio[i][j >> 1] = ((test[i*4+j+1]-test[i*4+j]) << 10) / test[i*4+j];
        stat[i] = canon_600_color (ratio[i], mar);
      }
      if ((st = stat[0] | stat[1]) > 1) goto next;
      for (i=0; i < 2; i++)
        if (stat[i])
          for (j=0; j < 2; j++)
            test[i*4+j*2+1] = test[i*4+j*2] * (0x400 + ratio[i][j]) >> 10;
      for (i=0; i < 8; i++)
        total[st][i] += test[i];
      count[st]++;
next: ;
    }
  if (count[0] | count[1]) {
    st = count[0]*200 < count[1];
    for (i=0; i < 4; i++)
      pre_mul[i] = 1.0 / (total[st][i] + total[st][i+4]);
  }
}

```

```
void CLASS canon_600_coeff()
{
  static const short table[6][12] = {
    { -190,702,-1878,2390,   1861,-1349,905,-393, -432,944,2617,-2105  },
    { -1203,1715,-1136,1648, 1388,-876,267,245,  -1641,2153,3921,-3409 },
    { -615,1127,-1563,2075,  1437,-925,509,3,     -756,1268,2519,-2007 },
    { -190,702,-1886,2398,   2153,-1641,763,-251, -452,964,3040,-2528  },
    { -190,702,-1878,2390,   1861,-1349,905,-393, -432,944,2617,-2105  },
    { -807,1319,-1785,2297,  1388,-876,769,-257,  -230,742,2067,-1555  } };
  int t=0, i, c;
  float mc, yc;

  mc = pre_mul[1] / pre_mul[2];
  yc = pre_mul[3] / pre_mul[2];
  if (mc > 1 && mc <= 1.28 && yc < 0.8789) t=1;
  if (mc > 1.28 && mc <= 2) {
    if  (yc < 0.8789) t=3;
    else if (yc <= 2) t=4;
  }
  if (flash_used) t=5;
  for (raw_color = i=0; i < 3; i++)
    FORCC rgb_cam[i][c] = table[t][i*4 + c] / 1024.0;
}

void CLASS canon_600_load_raw()
{
  uchar  data[1120], *dp;
  ushort *pix;
  int irow, row;

  for (irow=row=0; irow < height; irow++) {
    if (fread (data, 1, 1120, ifp) < 1120) derror();
    pix = raw_image + row*raw_width;
    for (dp=data; dp < data+1120;  dp+=10, pix+=8) {
      pix[0] = (dp[0] << 2) + (dp[1] >> 6    );
      pix[1] = (dp[2] << 2) + (dp[1] >> 4 & 3);
      pix[2] = (dp[3] << 2) + (dp[1] >> 2 & 3);
      pix[3] = (dp[4] << 2) + (dp[1]      & 3);
      pix[4] = (dp[5] << 2) + (dp[9]      & 3);
      pix[5] = (dp[6] << 2) + (dp[9] >> 2 & 3);
      pix[6] = (dp[7] << 2) + (dp[9] >> 4 & 3);
      pix[7] = (dp[8] << 2) + (dp[9] >> 6    );
    }
    if ((row+=2) > height) row = 1;
  }
}

void CLASS canon_600_correct()
{
  int row, col, val;
  static const short mul[4][2] =
  { { 1141,1145 }, { 1128,1109 }, { 1178,1149 }, { 1128,1109 } };

  for (row=0; row < height; row++)
    for (col=0; col < width; col++) {
      if ((val = BAYER(row,col) - black) < 0) val = 0;
      val = val * mul[row & 3][col & 1] >> 9;
      BAYER(row,col) = val;
    }
  canon_600_fixed_wb(1311);
  canon_600_auto_wb();
  canon_600_coeff();
  maximum = (0x3ff - black) * 1109 >> 9;
  black = 0;
}
```

```
581.
582. int CLASS canon_s2is()
583. {
584.   unsigned row;
585.
586.   for (row=0; row < 100; row++) {
587.     fseek (ifp, row*3340 + 3284, SEEK_SET);
588.     if (getc(ifp) > 15) return 1;
589.   }
590.   return 0;
591. }
592.
593. unsigned CLASS getbithuff (int nbits, ushort *huff)
594. {
595.   static unsigned bitbuf=0;
596.   static int vbits=0, reset=0;
597.   unsigned c;
598.
599.   if (nbits > 25) return 0;
600.   if (nbits < 0)
601.     return bitbuf = vbits = reset = 0;
602.   if (nbits == 0 || vbits < 0) return 0;
603.   while (!reset && vbits < nbits && (c = fgetc(ifp)) != EOF &&
604.     !(reset = zero_after_ff && c == 0xff && fgetc(ifp))) {
605.     bitbuf = (bitbuf << 8) + (uchar) c;
606.     vbits += 8;
607.   }
608.   c = bitbuf << (32-vbits) >> (32-nbits);
609.   if (huff) {
610.     vbits -= huff[c] >> 8;
611.     c = (uchar) huff[c];
612.   } else
613.     vbits -= nbits;
614.   if (vbits < 0) derror();
615.   return c;
616. }
617.
618. #define getbits(n) getbithuff(n,0)
619. #define gethuff(h) getbithuff(*h,h+1)
620.
621. /*
622.    Construct a decode tree according the specification in *source.
623.    The first 16 bytes specify how many codes should be 1-bit, 2-bit
624.    3-bit, etc.  Bytes after that are the leaf values.
625.
626.    For example, if the source is
627.
628.     { 0,1,4,2,3,1,2,0,0,0,0,0,0,0,0,0,
629.       0x04,0x03,0x05,0x06,0x02,0x07,0x01,0x08,0x09,0x00,0x0a,0x0b,0xff  },
630.
631.    then the code is
632.
633.         00              0x04
634.         010             0x03
635.         011             0x05
636.         100             0x06
637.         101             0x02
638.         1100            0x07
639.         1101            0x01
640.         11100           0x08
641.         11101           0x09
642.         11110           0x00
643.         111110          0x0a
644.         1111110         0x0b
645.         1111111         0xff
```

```
646.  */
647. ushort * CLASS make_decoder_ref (const uchar **source)
648. {
649.   int max, len, h, i, j;
650.   const uchar *count;
651.   ushort *huff;
652.
653.   count = (*source += 16) - 17;
654.   for (max=16; max && !count[max]; max--);
655.   huff = (ushort *) calloc (1 + (1 << max), sizeof *huff);
656.   merror (huff, "make_decoder()");
657.   huff[0] = max;
658.   for (h=len=1; len <= max; len++)
659.     for (i=0; i < count[len]; i++, ++*source)
660.       for (j=0; j < 1 << (max-len); j++)
661.         if (h <= 1 << max)
662.           huff[h++] = len << 8 | **source;
663.   return huff;
664. }
665.
666. ushort * CLASS make_decoder (const uchar *source)
667. {
668.   return make_decoder_ref (&source);
669. }
670.
671. void CLASS crw_init_tables (unsigned table, ushort *huff[2])
672. {
673.   static const uchar first_tree[3][29] = {
674.     { 0,1,4,2,3,1,2,0,0,0,0,0,0,0,0,
675.       0x04,0x03,0x05,0x06,0x02,0x07,0x01,0x08,0x09,0x00,0x0a,0x0b,0xff  },
676.     { 0,2,2,3,1,1,1,1,2,0,0,0,0,0,0,
677.       0x03,0x02,0x04,0x01,0x05,0x00,0x06,0x07,0x09,0x08,0x0a,0x0b,0xff  },
678.     { 0,0,6,3,1,1,2,0,0,0,0,0,0,0,0,
679.       0x06,0x05,0x07,0x04,0x08,0x03,0x09,0x02,0x00,0x0a,0x01,0x0b,0xff  },
680.   };
681.   static const uchar second_tree[3][180] = {
682.     { 0,2,2,2,1,4,2,1,2,5,1,1,0,0,0,139,
683.       0x03,0x04,0x02,0x05,0x01,0x06,0x07,0x08,
684.       0x12,0x13,0x11,0x14,0x09,0x15,0x22,0x00,0x21,0x16,0x0a,0xf0,
685.       0x23,0x17,0x24,0x31,0x32,0x18,0x19,0x33,0x25,0x41,0x34,0x42,
686.       0x35,0x51,0x36,0x37,0x38,0x29,0x79,0x26,0x1a,0x39,0x56,0x57,
687.       0x28,0x27,0x52,0x55,0x58,0x43,0x76,0x59,0x77,0x54,0x61,0xf9,
688.       0x71,0x78,0x75,0x96,0x97,0x49,0xb7,0x53,0xd7,0x74,0xb6,0x98,
689.       0x47,0x48,0x95,0x69,0x99,0x91,0xfa,0xb8,0x68,0xb5,0xb9,0xd6,
690.       0xf7,0xd8,0x67,0x46,0x45,0x94,0x89,0xf8,0x81,0xd5,0xf6,0xb4,
691.       0x88,0xb1,0x2a,0x44,0x72,0xd9,0x87,0x66,0xd4,0xf5,0x3a,0xa7,
692.       0x73,0xa9,0xa8,0x86,0x62,0xc7,0x65,0xc8,0xc9,0xa1,0xf4,0xd1,
693.       0xe9,0x5a,0x92,0x85,0xa6,0xe7,0x93,0xe8,0xc1,0xc6,0x7a,0x64,
694.       0xe1,0x4a,0x6a,0xe6,0xb3,0xf1,0xd3,0xa5,0x8a,0xb2,0x9a,0xba,
695.       0x84,0xa4,0x63,0xe5,0xc5,0xf3,0xd2,0xc4,0x82,0xaa,0xda,0xe4,
696.       0xf2,0xca,0x83,0xa3,0xa2,0xc3,0xea,0xc2,0xe2,0xe3,0xff,0xff  },
697.     { 0,2,2,1,4,1,4,1,3,3,1,0,0,0,0,140,
698.       0x02,0x03,0x01,0x04,0x05,0x12,0x11,0x06,
699.       0x13,0x07,0x08,0x14,0x22,0x09,0x21,0x00,0x23,0x15,0x31,0x32,
700.       0x0a,0x16,0xf0,0x24,0x33,0x41,0x42,0x19,0x17,0x25,0x18,0x51,
701.       0x34,0x43,0x52,0x29,0x35,0x61,0x39,0x71,0x62,0x36,0x53,0x26,
702.       0x38,0x1a,0x37,0x81,0x27,0x91,0x79,0x55,0x45,0x28,0x72,0x59,
703.       0xa1,0xb1,0x44,0x69,0x54,0x58,0xd1,0xfa,0x57,0xe1,0xf1,0xb9,
704.       0x49,0x47,0x63,0x6a,0xf9,0x56,0x46,0xa8,0x2a,0x4a,0x78,0x99,
705.       0x3a,0x75,0x74,0x86,0x65,0xc1,0x76,0xb6,0x96,0xd6,0x89,0x85,
706.       0xc9,0xf5,0x95,0xb4,0xc7,0xf7,0x8a,0x97,0xb8,0x73,0xb7,0xd8,
707.       0xd9,0x87,0xa7,0x7a,0x48,0x82,0x84,0xea,0xf4,0xa6,0xc5,0x5a,
708.       0x94,0xa4,0xc6,0x92,0xc3,0x68,0xb5,0xc8,0xe4,0xe5,0xe6,0xe9,
709.       0xa2,0xa3,0xe3,0xc2,0x66,0x67,0x93,0xaa,0xd4,0xd5,0xe7,0xf8,
710.       0x88,0x9a,0xd7,0x77,0xc4,0x64,0xe2,0x98,0xa5,0xca,0xda,0xe8,
```

```
      0xf3,0xf6,0xa9,0xb2,0xb3,0xf2,0xd2,0x83,0xba,0xd3,0xff,0xff  },
    { 0,0,6,2,1,3,3,2,5,1,2,2,8,10,0,117,
      0x04,0x05,0x03,0x06,0x02,0x07,0x01,0x08,
      0x09,0x12,0x13,0x14,0x11,0x15,0x0a,0x16,0x17,0xf0,0x00,0x22,
      0x21,0x18,0x23,0x19,0x24,0x32,0x31,0x25,0x33,0x38,0x37,0x34,
      0x35,0x36,0x39,0x79,0x57,0x58,0x59,0x28,0x56,0x78,0x27,0x41,
      0x29,0x77,0x26,0x42,0x76,0x99,0x1a,0x55,0x98,0x97,0xf9,0x48,
      0x54,0x96,0x89,0x47,0xb7,0x49,0xfa,0x75,0x68,0xb6,0x67,0x69,
      0xb9,0xb8,0xd8,0x52,0xd7,0x88,0xb5,0x74,0x51,0x46,0xd9,0xf8,
      0x3a,0xd6,0x87,0x45,0x7a,0x95,0xd5,0xf6,0x86,0xb4,0xa9,0x94,
      0x53,0x2a,0xa8,0x43,0xf5,0xf7,0xd4,0x66,0xa7,0x5a,0x44,0x8a,
      0xc9,0xe8,0xc8,0xe7,0x9a,0x6a,0x73,0x4a,0x61,0xc7,0xf4,0xc6,
      0x65,0xe9,0x72,0xe6,0x71,0x91,0x93,0xa6,0xda,0x92,0x85,0x62,
      0xf3,0xc5,0xb2,0xa4,0x84,0xba,0x64,0xa5,0xb3,0xd2,0x81,0xe5,
      0xd3,0xaa,0xc4,0xca,0xf2,0xb1,0xe4,0xd1,0x83,0x63,0xea,0xc3,
      0xe2,0x82,0xf1,0xa3,0xc2,0xa1,0xc1,0xe3,0xa2,0xe1,0xff,0xff  }
  };
  if (table > 2) table = 2;
  huff[0] = make_decoder ( first_tree[table]);
  huff[1] = make_decoder (second_tree[table]);
}

/*
   Return 0 if the image starts with compressed data,
   1 if it starts with uncompressed low-order bits.

   In Canon compressed data, 0xff is always followed by 0x00.
 */
int CLASS canon_has_lowbits()
{
  uchar test[0x4000];
  int ret=1, i;

  fseek (ifp, 0, SEEK_SET);
  fread (test, 1, sizeof test, ifp);
  for (i=540; i < sizeof test - 1; i++)
    if (test[i] == 0xff) {
      if (test[i+1]) return 1;
      ret=0;
    }
  return ret;
}

void CLASS canon_load_raw()
{
  ushort *pixel, *prow, *huff[2];
  int nblocks, lowbits, i, c, row, r, save, val;
  int block, diffbuf[64], leaf, len, diff, carry=0, pnum=0, base[2];

  crw_init_tables (tiff_compress, huff);
  lowbits = canon_has_lowbits();
  if (!lowbits) maximum = 0x3ff;
  fseek (ifp, 540 + lowbits*raw_height*raw_width/4, SEEK_SET);
  zero_after_ff = 1;
  getbits(-1);
  for (row=0; row < raw_height; row+=8) {
    pixel = raw_image + row*raw_width;
    nblocks = MIN (8, raw_height-row) * raw_width >> 6;
    for (block=0; block < nblocks; block++) {
      memset (diffbuf, 0, sizeof diffbuf);
      for (i=0; i < 64; i++ ) {
        leaf = gethuff(huff[i > 0]);
        if (leaf == 0 && i) break;
        if (leaf == 0xff) continue;
        i  += leaf >> 4;
```

```
776.          len = leaf & 15;
777.          if (len == 0) continue;
778.          diff = getbits(len);
779.          if ((diff & (1 << (len-1))) == 0)
780.            diff -= (1 << len) - 1;
781.          if (i < 64) diffbuf[i] = diff;
782.        }
783.        diffbuf[0] += carry;
784.        carry = diffbuf[0];
785.        for (i=0; i < 64; i++ ) {
786.          if (pnum++ % raw_width == 0)
787.            base[0] = base[1] = 512;
788.          if ((pixel[(block << 6) + i] = base[i & 1] += diffbuf[i]) >> 10)
789.            derror();
790.        }
791.      }
792.      if (lowbits) {
793.        save = ftell(ifp);
794.        fseek (ifp, 26 + row*raw_width/4, SEEK_SET);
795.        for (prow=pixel, i=0; i < raw_width*2; i++) {
796.          c = fgetc(ifp);
797.          for (r=0; r < 8; r+=2, prow++) {
798.            val = (*prow << 2) + ((c >> r) & 3);
799.            if (raw_width == 2672 && val < 512) val += 2;
800.            *prow = val;
801.          }
802.        }
803.        fseek (ifp, save, SEEK_SET);
804.      }
805.    }
806.    FORC(2) free (huff[c]);
807. }
808.
809. struct jhead {
810.    int algo, bits, high, wide, clrs, sraw, psv, restart, vpred[6];
811.    ushort quant[64], idct[64], *huff[20], *free[20], *row;
812. };
813.
814. int CLASS ljpeg_start (struct jhead *jh, int info_only)
815. {
816.    ushort c, tag, len;
817.    uchar data[0x10000];
818.    const uchar *dp;
819.
820.    memset (jh, 0, sizeof *jh);
821.    jh->restart = INT_MAX;
822.    if ((fgetc(ifp),fgetc(ifp)) != 0xd8) return 0;
823.    do {
824.      if (!fread (data, 2, 2, ifp)) return 0;
825.      tag =  data[0] << 8 | data[1];
826.      len = (data[2] << 8 | data[3]) - 2;
827.      if (tag <= 0xff00) return 0;
828.      fread (data, 1, len, ifp);
829.      switch (tag) {
830.        case 0xffc3:
831.          jh->sraw = ((data[7] >> 4) * (data[7] & 15) - 1) & 3;
832.        case 0xffc1:
833.        case 0xffc0:
834.          jh->algo = tag & 0xff;
835.          jh->bits = data[0];
836.          jh->high = data[1] << 8 | data[2];
837.          jh->wide = data[3] << 8 | data[4];
838.          jh->clrs = data[5] + jh->sraw;
839.          if (len == 9 && !dng_version) getc(ifp);
840.          break;
```

```
841.       case 0xffc4:
842.         if (info_only) break;
843.         for (dp = data; dp < data+len && !((c = *dp++) & -20); )
844.           jh->free[c] = jh->huff[c] = make_decoder_ref (&dp);
845.         break;
846.       case 0xffda:
847.         jh->psv = data[1+data[0]*2];
848.         jh->bits -= data[3+data[0]*2] & 15;
849.         break;
850.       case 0xffdb:
851.         FORC(64) jh->quant[c] = data[c*2+1] << 8 | data[c*2+2];
852.         break;
853.       case 0xffdd:
854.         jh->restart = data[0] << 8 | data[1];
855.     }
856.   } while (tag != 0xffda);
857.   if (jh->bits > 16 || jh->clrs > 6 ||
858.      !jh->bits || !jh->high || !jh->wide || !jh->clrs) return 0;
859.   if (info_only) return 1;
860.   if (!jh->huff[0]) return 0;
861.   FORC(19) if (!jh->huff[c+1]) jh->huff[c+1] = jh->huff[c];
862.   if (jh->sraw) {
863.     FORC(4)        jh->huff[2+c] = jh->huff[1];
864.     FORC(jh->sraw) jh->huff[1+c] = jh->huff[0];
865.   }
866.   jh->row = (ushort *) calloc (jh->wide*jh->clrs, 4);
867.   merror (jh->row, "ljpeg_start()");
868.   return zero_after_ff = 1;
869. }
870.
871. void CLASS ljpeg_end (struct jhead *jh)
872. {
873.   int c;
874.   FORC4 if (jh->free[c]) free (jh->free[c]);
875.   free (jh->row);
876. }
877.
878. int CLASS ljpeg_diff (ushort *huff)
879. {
880.   int len, diff;
881.
882.   len = gethuff(huff);
883.   if (len == 16 && (!dng_version || dng_version >= 0x1010000))
884.     return -32768;
885.   diff = getbits(len);
886.   if ((diff & (1 << (len-1))) == 0)
887.     diff -= (1 << len) - 1;
888.   return diff;
889. }
890.
891. ushort * CLASS ljpeg_row (int jrow, struct jhead *jh)
892. {
893.   int col, c, diff, pred, spred=0;
894.   ushort mark=0, *row[3];
895.
896.   if (jrow * jh->wide % jh->restart == 0) {
897.     FORC(6) jh->vpred[c] = 1 << (jh->bits-1);
898.     if (jrow) {
899.       fseek (ifp, -2, SEEK_CUR);
900.       do mark = (mark << 8) + (c = fgetc(ifp));
901.       while (c != EOF && mark >> 4 != 0xffd);
902.     }
903.     getbits(-1);
904.   }
905.   FORC3 row[c] = jh->row + jh->wide*jh->clrs*((jrow+c) & 1);
```

```
906.   for (col=0; col < jh->wide; col++)
907.     FORC(jh->clrs) {
908.       diff = ljpeg_diff (jh->huff[c]);
909.       if (jh->sraw && c <= jh->sraw && (col | c))
910.                     pred = spred;
911.       else if (col) pred = row[0][-jh->clrs];
912.       else          pred = (jh->vpred[c] += diff) - diff;
913.       if (jrow && col) switch (jh->psv) {
914.         case 1: break;
915.         case 2: pred = row[1][0];                                          break;
916.         case 3: pred = row[1][-jh->clrs];                                  break;
917.         case 4: pred = pred +   row[1][0] - row[1][-jh->clrs];             break;
918.         case 5: pred = pred + ((row[1][0] - row[1][-jh->clrs]) >> 1);      break;
919.         case 6: pred = row[1][0] + ((pred - row[1][-jh->clrs]) >> 1);      break;
920.         case 7: pred = (pred + row[1][0]) >> 1;                            break;
921.         default: pred = 0;
922.       }
923.       if ((**row = pred + diff) >> jh->bits) derror();
924.       if (c <= jh->sraw) spred = **row;
925.       row[0]++; row[1]++;
926.     }
927.   return row[2];
928. }
929.
930. void CLASS lossless_jpeg_load_raw()
931. {
932.   int jwide, jrow, jcol, val, jidx, i, j, row=0, col=0;
933.   struct jhead jh;
934.   ushort *rp;
935.
936.   if (!ljpeg_start (&jh, 0)) return;
937.   jwide = jh.wide * jh.clrs;
938.
939.   for (jrow=0; jrow < jh.high; jrow++) {
940.     rp = ljpeg_row (jrow, &jh);
941.     if (load_flags & 1)
942.       row = jrow & 1 ? height-1-jrow/2 : jrow/2;
943.     for (jcol=0; jcol < jwide; jcol++) {
944.       val = curve[*rp++];
945.       if (cr2_slice[0]) {
946.         jidx = jrow*jwide + jcol;
947.         i = jidx / (cr2_slice[1]*raw_height);
948.         if ((j = i >= cr2_slice[0]))
949.                   i  = cr2_slice[0];
950.         jidx -= i * (cr2_slice[1]*raw_height);
951.         row = jidx / cr2_slice[1+j];
952.         col = jidx % cr2_slice[1+j] + i*cr2_slice[1];
953.       }
954.       if (raw_width == 3984 && (col -= 2) < 0)
955.         col += (row--,raw_width);
956.       if ((unsigned) row < raw_height) RAW(row,col) = val;
957.       if (++col >= raw_width)
958.         col = (row++,0);
959.     }
960.   }
961.   ljpeg_end (&jh);
962. }
963.
964. void CLASS canon_sraw_load_raw()
965. {
966.   struct jhead jh;
967.   short *rp=0, (*ip)[4];
968.   int jwide, slice, scol, ecol, row, col, jrow=0, jcol=0, pix[3], c;
969.   int v[3]={0,0,0}, ver, hue;
970.   char *cp;
```

```
971.
972.    if (!ljpeg_start (&jh, 0) || jh.clrs < 4) return;
973.    jwide = (jh.wide >>= 1) * jh.clrs;
974.
975.    for (ecol=slice=0; slice <= cr2_slice[0]; slice++) {
976.      scol = ecol;
977.      ecol += cr2_slice[1] * 2 / jh.clrs;
978.      if (!cr2_slice[0] || ecol > raw_width-1) ecol = raw_width & -2;
979.      for (row=0; row < height; row += (jh.clrs >> 1) - 1) {
980.        ip = (short (*)[4]) image + row*width;
981.        for (col=scol; col < ecol; col+=2, jcol+=jh.clrs) {
982.          if ((jcol %= jwide) == 0)
983.            rp = (short *) ljpeg_row (jrow++, &jh);
984.          if (col >= width) continue;
985.          FORC (jh.clrs-2)
986.            ip[col + (c >> 1)*width + (c & 1)][0] = rp[jcol+c];
987.          ip[col][1] = rp[jcol+jh.clrs-2] - 16384;
988.          ip[col][2] = rp[jcol+jh.clrs-1] - 16384;
989.        }
990.      }
991.    }
992.    for (cp=model2; *cp && !isdigit(*cp); cp++);
993.    sscanf (cp, "%d.%d.%d", v, v+1, v+2);
994.    ver = (v[0]*1000 + v[1])*1000 + v[2];
995.    hue = (jh.sraw+1) << 2;
996.    if (unique_id >= 0x80000281 || (unique_id == 0x80000218 && ver > 1000006))
997.      hue = jh.sraw << 1;
998.    ip = (short (*)[4]) image;
999.    rp = ip[0];
1000.   for (row=0; row < height; row++, ip+=width) {
1001.     if (row & (jh.sraw >> 1))
1002.       for (col=0; col < width; col+=2)
1003.         for (c=1; c < 3; c++)
1004.           if (row == height-1)
1005.                ip[col][c] =  ip[col-width][c];
1006.           else ip[col][c] = (ip[col-width][c] + ip[col+width][c] + 1) >> 1;
1007.     for (col=1; col < width; col+=2)
1008.       for (c=1; c < 3; c++)
1009.         if (col == width-1)
1010.              ip[col][c] =  ip[col-1][c];
1011.         else ip[col][c] = (ip[col-1][c] + ip[col+1][c] + 1) >> 1;
1012.   }
1013.   for ( ; rp < ip[0]; rp+=4) {
1014.     if (unique_id == 0x80000218 ||
1015.         unique_id == 0x80000250 ||
1016.         unique_id == 0x80000261 ||
1017.         unique_id == 0x80000281 ||
1018.         unique_id == 0x80000287) {
1019.       rp[1] = (rp[1] << 2) + hue;
1020.       rp[2] = (rp[2] << 2) + hue;
1021.       pix[0] = rp[0] + ((   50*rp[1] + 22929*rp[2]) >> 14);
1022.       pix[1] = rp[0] + ((-5640*rp[1] - 11751*rp[2]) >> 14);
1023.       pix[2] = rp[0] + ((29040*rp[1] -   101*rp[2]) >> 14);
1024.     } else {
1025.       if (unique_id < 0x80000218) rp[0] -= 512;
1026.       pix[0] = rp[0] + rp[2];
1027.       pix[2] = rp[0] + rp[1];
1028.       pix[1] = rp[0] + ((-778*rp[1] - (rp[2] << 11)) >> 12);
1029.     }
1030.     FORC3 rp[c] = CLIP(pix[c] * sraw_mul[c] >> 10);
1031.   }
1032.   ljpeg_end (&jh);
1033.   maximum = 0x3fff;
1034. }
1035.
```

```
1036.void CLASS adobe_copy_pixel (unsigned row, unsigned col, ushort **rp)
1037.{
1038.  int c;
1039.
1040.  if (tiff_samples == 2 && shot_select) (*rp)++;
1041.  if (raw_image) {
1042.    if (row < raw_height && col < raw_width)
1043.      RAW(row,col) = curve[**rp];
1044.    *rp += tiff_samples;
1045.  } else {
1046.    if (row < height && col < width)
1047.      FORC(tiff_samples)
1048.        image[row*width+col][c] = curve[(*rp)[c]];
1049.    *rp += tiff_samples;
1050.  }
1051.  if (tiff_samples == 2 && shot_select) (*rp)--;
1052.}
1053.
1054.void CLASS ljpeg_idct (struct jhead *jh)
1055.{
1056.  int c, i, j, len, skip, coef;
1057.  float work[3][8][8];
1058.  static float cs[106] = { 0 };
1059.  static const uchar zigzag[80] =
1060.  {  0, 1, 8,16, 9, 2, 3,10,17,24,32,25,18,11, 4, 5,12,19,26,33,
1061.    40,48,41,34,27,20,13, 6, 7,14,21,28,35,42,49,56,57,50,43,36,
1062.    29,22,15,23,30,37,44,51,58,59,52,45,38,31,39,46,53,60,61,54,
1063.    47,55,62,63,63,63,63,63,63,63,63,63,63,63,63,63,63,63,63,63 };
1064.
1065.  if (!cs[0])
1066.    FORC(106) cs[c] = cos((c & 31)*M_PI/16)/2;
1067.  memset (work, 0, sizeof work);
1068.  work[0][0][0] = jh->vpred[0] += ljpeg_diff (jh->huff[0]) * jh->quant[0];
1069.  for (i=1; i < 64; i++ ) {
1070.    len = gethuff (jh->huff[16]);
1071.    i += skip = len >> 4;
1072.    if (!(len &= 15) && skip < 15) break;
1073.    coef = getbits(len);
1074.    if ((coef & (1 << (len-1))) == 0)
1075.      coef -= (1 << len) - 1;
1076.    ((float *)work)[zigzag[i]] = coef * jh->quant[i];
1077.  }
1078.  FORC(8) work[0][0][c] *= M_SQRT1_2;
1079.  FORC(8) work[0][c][0] *= M_SQRT1_2;
1080.  for (i=0; i < 8; i++)
1081.    for (j=0; j < 8; j++)
1082.      FORC(8) work[1][i][j] += work[0][i][c] * cs[(j*2+1)*c];
1083.  for (i=0; i < 8; i++)
1084.    for (j=0; j < 8; j++)
1085.      FORC(8) work[2][i][j] += work[1][c][j] * cs[(i*2+1)*c];
1086.
1087.  FORC(64) jh->idct[c] = CLIP(((float *)work[2])[c]+0.5);
1088.}
1089.
1090.void CLASS lossless_dng_load_raw()
1091.{
1092.  unsigned save, trow=0, tcol=0, jwide, jrow, jcol, row, col, i, j;
1093.  struct jhead jh;
1094.  ushort *rp;
1095.
1096.  while (trow < raw_height) {
1097.    save = ftell(ifp);
1098.    if (tile_length < INT_MAX)
1099.      fseek (ifp, get4(), SEEK_SET);
1100.    if (!ljpeg_start (&jh, 0)) break;
```

```
1101.    jwide = jh.wide;
1102.    if (filters) jwide *= jh.clrs;
1103.    jwide /= MIN (is_raw, tiff_samples);
1104.    switch (jh.algo) {
1105.      case 0xc1:
1106.        jh.vpred[0] = 16384;
1107.        getbits(-1);
1108.        for (jrow=0; jrow+7 < jh.high; jrow += 8) {
1109.          for (jcol=0; jcol+7 < jh.wide; jcol += 8) {
1110.            ljpeg_idct (&jh);
1111.            rp = jh.idct;
1112.            row = trow + jcol/tile_width + jrow*2;
1113.            col = tcol + jcol%tile_width;
1114.            for (i=0; i < 16; i+=2)
1115.              for (j=0; j < 8; j++)
1116.                adobe_copy_pixel (row+i, col+j, &rp);
1117.          }
1118.        }
1119.        break;
1120.      case 0xc3:
1121.        for (row=col=jrow=0; jrow < jh.high; jrow++) {
1122.          rp = ljpeg_row (jrow, &jh);
1123.          for (jcol=0; jcol < jwide; jcol++) {
1124.            adobe_copy_pixel (trow+row, tcol+col, &rp);
1125.            if (++col >= tile_width || col >= raw_width)
1126.              row += 1 + (col = 0);
1127.          }
1128.        }
1129.    }
1130.    fseek (ifp, save+4, SEEK_SET);
1131.    if ((tcol += tile_width) >= raw_width)
1132.      trow += tile_length + (tcol = 0);
1133.    ljpeg_end (&jh);
1134.  }
1135.}
1136.
1137.void CLASS packed_dng_load_raw()
1138.{
1139.  ushort *pixel, *rp;
1140.  int row, col;
1141.
1142.  pixel = (ushort *) calloc (raw_width, tiff_samples*sizeof *pixel);
1143.  merror (pixel, "packed_dng_load_raw()");
1144.  for (row=0; row < raw_height; row++) {
1145.    if (tiff_bps == 16)
1146.      read_shorts (pixel, raw_width * tiff_samples);
1147.    else {
1148.      getbits(-1);
1149.      for (col=0; col < raw_width * tiff_samples; col++)
1150.        pixel[col] = getbits(tiff_bps);
1151.    }
1152.    for (rp=pixel, col=0; col < raw_width; col++)
1153.      adobe_copy_pixel (row, col, &rp);
1154.  }
1155.  free (pixel);
1156.}
1157.
1158.void CLASS pentax_load_raw()
1159.{
1160.  ushort bit[2][15], huff[4097];
1161.  int dep, row, col, diff, c, i;
1162.  ushort vpred[2][2] = {{0,0},{0,0}}, hpred[2];
1163.
1164.  fseek (ifp, meta_offset, SEEK_SET);
1165.  dep = (get2() + 12) & 15;
```

```
  fseek (ifp, 12, SEEK_CUR);
  FORC(dep) bit[0][c] = get2();
  FORC(dep) bit[1][c] = fgetc(ifp);
  FORC(dep)
    for (i=bit[0][c]; i <= ((bit[0][c]+(4096 >> bit[1][c])-1) & 4095); )
      huff[++i] = bit[1][c] << 8 | c;
  huff[0] = 12;
  fseek (ifp, data_offset, SEEK_SET);
  getbits(-1);
  for (row=0; row < raw_height; row++)
    for (col=0; col < raw_width; col++) {
      diff = ljpeg_diff (huff);
      if (col < 2) hpred[col] = vpred[row & 1][col] += diff;
      else         hpred[col & 1] += diff;
      RAW(row,col) = hpred[col & 1];
      if (hpred[col & 1] >> tiff_bps) derror();
    }
}

void CLASS nikon_load_raw()
{
  static const uchar nikon_tree[][32] = {
    { 0,1,5,1,1,1,1,1,1,2,0,0,0,0,0,0,  /* 12-bit lossy */
      5,4,3,6,2,7,1,0,8,9,11,10,12 },
    { 0,1,5,1,1,1,1,1,1,2,0,0,0,0,0,0,  /* 12-bit lossy after split */
      0x39,0x5a,0x38,0x27,0x16,5,4,3,2,1,0,11,12,12 },
    { 0,1,4,2,3,1,2,0,0,0,0,0,0,0,0,0,  /* 12-bit lossless */
      5,4,6,3,7,2,8,1,9,0,10,11,12 },
    { 0,1,4,3,1,1,1,1,1,2,0,0,0,0,0,0,  /* 14-bit lossy */
      5,6,4,7,8,3,9,2,1,0,10,11,12,13,14 },
    { 0,1,5,1,1,1,1,1,1,1,2,0,0,0,0,0,  /* 14-bit lossy after split */
      8,0x5c,0x4b,0x3a,0x29,7,6,5,4,3,2,1,0,13,14 },
    { 0,1,4,2,2,3,1,2,0,0,0,0,0,0,0,0,  /* 14-bit lossless */
      7,6,8,5,9,4,10,3,11,12,2,0,1,13,14 } };
  ushort *huff, ver0, ver1, vpred[2][2], hpred[2], csize;
  int i, min, max, step=0, tree=0, split=0, row, col, len, shl, diff;

  fseek (ifp, meta_offset, SEEK_SET);
  ver0 = fgetc(ifp);
  ver1 = fgetc(ifp);
  if (ver0 == 0x49 || ver1 == 0x58)
    fseek (ifp, 2110, SEEK_CUR);
  if (ver0 == 0x46) tree = 2;
  if (tiff_bps == 14) tree += 3;
  read_shorts (vpred[0], 4);
  max = 1 << tiff_bps & 0x7fff;
  if ((csize = get2()) > 1)
    step = max / (csize-1);
  if (ver0 == 0x44 && ver1 == 0x20 && step > 0) {
    for (i=0; i < csize; i++)
      curve[i*step] = get2();
    for (i=0; i < max; i++)
      curve[i] = ( curve[i-i%step]*(step-i%step) +
                   curve[i-i%step+step]*(i%step) ) / step;
    fseek (ifp, meta_offset+562, SEEK_SET);
    split = get2();
  } else if (ver0 != 0x46 && csize <= 0x4001)
    read_shorts (curve, max=csize);
  while (curve[max-2] == curve[max-1]) max--;
  huff = make_decoder (nikon_tree[tree]);
  fseek (ifp, data_offset, SEEK_SET);
  getbits(-1);
  for (min=row=0; row < height; row++) {
    if (split && row == split) {
      free (huff);
```

```
1231.       huff = make_decoder (nikon_tree[tree+1]);
1232.       max += (min = 16) << 1;
1233.     }
1234.     for (col=0; col < raw_width; col++) {
1235.       i = gethuff(huff);
1236.       len = i & 15;
1237.       shl = i >> 4;
1238.       diff = ((getbits(len-shl) << 1) + 1) << shl >> 1;
1239.       if ((diff & (1 << (len-1))) == 0)
1240.         diff -= (1 << len) - !shl;
1241.       if (col < 2) hpred[col] = vpred[row & 1][col] += diff;
1242.       else         hpred[col & 1] += diff;
1243.       if ((ushort)(hpred[col & 1] + min) >= max) derror();
1244.       RAW(row,col) = curve[LIM((short)hpred[col & 1],0,0x3fff)];
1245.     }
1246.   }
1247.   free (huff);
1248. }
1249.
1250. void CLASS nikon_yuv_load_raw()
1251. {
1252.   int row, col, yuv[4], rgb[3], b, c;
1253.   UINT64 bitbuf=0;
1254.
1255.   for (row=0; row < raw_height; row++)
1256.     for (col=0; col < raw_width; col++) {
1257.       if (!(b = col & 1)) {
1258.         bitbuf = 0;
1259.         FORC(6) bitbuf |= (UINT64) fgetc(ifp) << c*8;
1260.         FORC(4) yuv[c] = (bitbuf >> c*12 & 0xfff) - (c >> 1 << 11);
1261.       }
1262.       rgb[0] = yuv[b] + 1.370705*yuv[3];
1263.       rgb[1] = yuv[b] - 0.337633*yuv[2] - 0.698001*yuv[3];
1264.       rgb[2] = yuv[b] + 1.732446*yuv[2];
1265.       FORC3 image[row*width+col][c] = curve[LIM(rgb[c],0,0xfff)] / cam_mul[c];
1266.     }
1267. }
1268.
1269. /*
1270.    Returns 1 for a Coolpix 995, 0 for anything else.
1271.  */
1272. int CLASS nikon_e995()
1273. {
1274.   int i, histo[256];
1275.   const uchar often[] = { 0x00, 0x55, 0xaa, 0xff };
1276.
1277.   memset (histo, 0, sizeof histo);
1278.   fseek (ifp, -2000, SEEK_END);
1279.   for (i=0; i < 2000; i++)
1280.     histo[fgetc(ifp)]++;
1281.   for (i=0; i < 4; i++)
1282.     if (histo[often[i]] < 200)
1283.       return 0;
1284.   return 1;
1285. }
1286.
1287. /*
1288.    Returns 1 for a Coolpix 2100, 0 for anything else.
1289.  */
1290. int CLASS nikon_e2100()
1291. {
1292.   uchar t[12];
1293.   int i;
1294.
1295.   fseek (ifp, 0, SEEK_SET);
```

```
  for (i=0; i < 1024; i++) {
    fread (t, 1, 12, ifp);
    if (((t[2] & t[4] & t[7] & t[9]) >> 4
        & t[1] & t[6] & t[8] & t[11] & 3) != 3)
      return 0;
  }
  return 1;
}

void CLASS nikon_3700()
{
  int bits, i;
  uchar dp[24];
  static const struct {
    int bits;
    char make[12], model[15];
  } table[] = {
    { 0x00, "Pentax",  "Optio 33WR" },
    { 0x03, "Nikon",   "E3200" },
    { 0x32, "Nikon",   "E3700" },
    { 0x33, "Olympus", "C740UZ" } };

  fseek (ifp, 3072, SEEK_SET);
  fread (dp, 1, 24, ifp);
  bits = (dp[8] & 3) << 4 | (dp[20] & 3);
  for (i=0; i < sizeof table / sizeof *table; i++)
    if (bits == table[i].bits) {
      strcpy (make,  table[i].make );
      strcpy (model, table[i].model);
    }
}

/*
   Separates a Minolta DiMAGE Z2 from a Nikon E4300.
 */
int CLASS minolta_z2()
{
  int i, nz;
  char tail[424];

  fseek (ifp, -sizeof tail, SEEK_END);
  fread (tail, 1, sizeof tail, ifp);
  for (nz=i=0; i < sizeof tail; i++)
    if (tail[i]) nz++;
  return nz > 20;
}

void CLASS jpeg_thumb();

void CLASS ppm_thumb()
{
  char *thumb;
  thumb_length = thumb_width*thumb_height*3;
  thumb = (char *) malloc (thumb_length);
  merror (thumb, "ppm_thumb()");
  fprintf (ofp, "P6\n%d %d\n255\n", thumb_width, thumb_height);
  fread  (thumb, 1, thumb_length, ifp);
  fwrite (thumb, 1, thumb_length, ofp);
  free (thumb);
}

void CLASS ppm16_thumb()
{
  int i;
  char *thumb;
```

```
1361.  thumb_length = thumb_width*thumb_height*3;
1362.  thumb = (char *) calloc (thumb_length, 2);
1363.  merror (thumb, "ppm16_thumb()");
1364.  read_shorts ((ushort *) thumb, thumb_length);
1365.  for (i=0; i < thumb_length; i++)
1366.    thumb[i] = ((ushort *) thumb)[i] >> 8;
1367.  fprintf (ofp, "P6\n%d %d\n255\n", thumb_width, thumb_height);
1368.  fwrite (thumb, 1, thumb_length, ofp);
1369.  free (thumb);
1370.}
1371.
1372.void CLASS layer_thumb()
1373.{
1374.  int i, c;
1375.  char *thumb, map[][4] = { "012","102" };
1376.
1377.  colors = thumb_misc >> 5 & 7;
1378.  thumb_length = thumb_width*thumb_height;
1379.  thumb = (char *) calloc (colors, thumb_length);
1380.  merror (thumb, "layer_thumb()");
1381.  fprintf (ofp, "P%d\n%d %d\n255\n",
1382.        5 + (colors >> 1), thumb_width, thumb_height);
1383.  fread (thumb, thumb_length, colors, ifp);
1384.  for (i=0; i < thumb_length; i++)
1385.    FORCC putc (thumb[i+thumb_length*(map[thumb_misc >> 8][c]-'0')], ofp);
1386.  free (thumb);
1387.}
1388.
1389.void CLASS rollei_thumb()
1390.{
1391.  unsigned i;
1392.  ushort *thumb;
1393.
1394.  thumb_length = thumb_width * thumb_height;
1395.  thumb = (ushort *) calloc (thumb_length, 2);
1396.  merror (thumb, "rollei_thumb()");
1397.  fprintf (ofp, "P6\n%d %d\n255\n", thumb_width, thumb_height);
1398.  read_shorts (thumb, thumb_length);
1399.  for (i=0; i < thumb_length; i++) {
1400.    putc (thumb[i] << 3, ofp);
1401.    putc (thumb[i] >> 5  << 2, ofp);
1402.    putc (thumb[i] >> 11 << 3, ofp);
1403.  }
1404.  free (thumb);
1405.}
1406.
1407.void CLASS rollei_load_raw()
1408.{
1409.  uchar pixel[10];
1410.  unsigned iten=0, isix, i, buffer=0, todo[16];
1411.
1412.  isix = raw_width * raw_height * 5 / 8;
1413.  while (fread (pixel, 1, 10, ifp) == 10) {
1414.    for (i=0; i < 10; i+=2) {
1415.      todo[i]   = iten++;
1416.      todo[i+1] = pixel[i] << 8 | pixel[i+1];
1417.      buffer    = pixel[i] >> 2 | buffer << 6;
1418.    }
1419.    for (   ; i < 16; i+=2) {
1420.      todo[i]   = isix++;
1421.      todo[i+1] = buffer >> (14-i)*5;
1422.    }
1423.    for (i=0; i < 16; i+=2)
1424.      raw_image[todo[i]] = (todo[i+1] & 0x3ff);
1425.  }
```

```
  maximum = 0x3ff;
}

int CLASS raw (unsigned row, unsigned col)
{
  return (row < raw_height && col < raw_width) ? RAW(row,col) : 0;
}

void CLASS phase_one_flat_field (int is_float, int nc)
{
  ushort head[8];
  unsigned wide, high, y, x, c, rend, cend, row, col;
  float *mrow, num, mult[4];

  read_shorts (head, 8);
  if (head[2] * head[3] * head[4] * head[5] == 0) return;
  wide = head[2] / head[4] + (head[2] % head[4] != 0);
  high = head[3] / head[5] + (head[3] % head[5] != 0);
  mrow = (float *) calloc (nc*wide, sizeof *mrow);
  merror (mrow, "phase_one_flat_field()");
  for (y=0; y < high; y++) {
    for (x=0; x < wide; x++)
      for (c=0; c < nc; c+=2) {
        num = is_float ? getreal(11) : get2()/32768.0;
        if (y==0) mrow[c*wide+x] = num;
        else mrow[(c+1)*wide+x] = (num - mrow[c*wide+x]) / head[5];
      }
    if (y==0) continue;
    rend = head[1] + y*head[5];
    for (row = rend-head[5];
         row < raw_height && row < rend &&
         row < head[1]+head[3]-head[5]; row++) {
      for (x=1; x < wide; x++) {
        for (c=0; c < nc; c+=2) {
          mult[c] = mrow[c*wide+x-1];
          mult[c+1] = (mrow[c*wide+x] - mult[c]) / head[4];
        }
        cend = head[0] + x*head[4];
        for (col = cend-head[4];
             col < raw_width &&
             col < cend && col < head[0]+head[2]-head[4]; col++) {
          c = nc > 2 ? FC(row-top_margin,col-left_margin) : 0;
          if (!(c & 1)) {
            c = RAW(row,col) * mult[c];
            RAW(row,col) = LIM(c,0,65535);
          }
          for (c=0; c < nc; c+=2)
            mult[c] += mult[c+1];
        }
      }
      for (x=0; x < wide; x++)
        for (c=0; c < nc; c+=2)
          mrow[c*wide+x] += mrow[(c+1)*wide+x];
    }
  }
  free (mrow);
}

void CLASS phase_one_correct()
{
  unsigned entries, tag, data, save, col, row, type;
  int len, i, j, k, cip, val[4], dev[4], sum, max;
  int head[9], diff, mindiff=INT_MAX, off_412=0;
  static const signed char dir[12][2] =
    { {-1,-1}, {-1,1}, {1,-1}, {1,1}, {-2,0}, {0,-2}, {0,2}, {2,0},
```

```
    {-2,-2}, {-2,2}, {2,-2}, {2,2} };
float poly[8], num, cfrac, frac, mult[2], *yval[2];
ushort *xval[2];
int qmult_applied = 0, qlin_applied = 0;

if (half_size || !meta_length) return;
if (verbose) fprintf (stderr,_("Phase One correction...\n"));
fseek (ifp, meta_offset, SEEK_SET);
order = get2();
fseek (ifp, 6, SEEK_CUR);
fseek (ifp, meta_offset+get4(), SEEK_SET);
entries = get4();  get4();
while (entries--) {
  tag  = get4();
  len  = get4();
  data = get4();
  save = ftell(ifp);
  fseek (ifp, meta_offset+data, SEEK_SET);
  if (tag == 0x419) {                         /* Polynomial curve */
    for (get4(), i=0; i < 8; i++)
      poly[i] = getreal(11);
    poly[3] += (ph1.tag_210 - poly[7]) * poly[6] + 1;
    for (i=0; i < 0x10000; i++) {
      num = (poly[5]*i + poly[3])*i + poly[1];
      curve[i] = LIM(num,0,65535);
    } goto apply;                             /* apply to right half */
  } else if (tag == 0x41a) {                  /* Polynomial curve */
    for (i=0; i < 4; i++)
      poly[i] = getreal(11);
    for (i=0; i < 0x10000; i++) {
      for (num=0, j=4; j--; )
        num = num * i + poly[j];
      curve[i] = LIM(num+i,0,65535);
    } apply:                                  /* apply to whole image */
    for (row=0; row < raw_height; row++)
      for (col = (tag & 1)*ph1.split_col; col < raw_width; col++)
        RAW(row,col) = curve[RAW(row,col)];
  } else if (tag == 0x400) {                  /* Sensor defects */
    while ((len -= 8) >= 0) {
      col  = get2();
      row  = get2();
      type = get2(); get2();
      if (col >= raw_width) continue;
      if (type == 131 || type == 137)         /* Bad column */
        for (row=0; row < raw_height; row++)
          if (FC(row-top_margin,col-left_margin) == 1) {
            for (sum=i=0; i < 4; i++)
              sum += val[i] = raw (row+dir[i][0], col+dir[i][1]);
            for (max=i=0; i < 4; i++) {
              dev[i] = abs((val[i] << 2) - sum);
              if (dev[max] < dev[i]) max = i;
            }
            RAW(row,col) = (sum - val[max])/3.0 + 0.5;
          } else {
            for (sum=0, i=8; i < 12; i++)
              sum += raw (row+dir[i][0], col+dir[i][1]);
            RAW(row,col) = 0.5 + sum * 0.0732233 +
              (raw(row,col-2) + raw(row,col+2)) * 0.3535534;
          }
      else if (type == 129) {                 /* Bad pixel */
        if (row >= raw_height) continue;
        j = (FC(row-top_margin,col-left_margin) != 1) * 4;
        for (sum=0, i=j; i < j+8; i++)
          sum += raw (row+dir[i][0], col+dir[i][1]);
        RAW(row,col) = (sum + 4) >> 3;
```

```
1556.       }
1557.     }
1558.   } else if (tag == 0x401) {                  /* All-color flat fields */
1559.     phase_one_flat_field (1, 2);
1560.   } else if (tag == 0x416 || tag == 0x410) {
1561.     phase_one_flat_field (0, 2);
1562.   } else if (tag == 0x40b) {                  /* Red+blue flat field */
1563.     phase_one_flat_field (0, 4);
1564.   } else if (tag == 0x412) {
1565.     fseek (ifp, 36, SEEK_CUR);
1566.     diff = abs (get2() - ph1.tag_21a);
1567.     if (mindiff > diff) {
1568.       mindiff = diff;
1569.       off_412 = ftell(ifp) - 38;
1570.     }
1571.   } else if (tag == 0x41f && !qlin_applied) { /* Quadrant linearization */
1572.     ushort lc[2][2][16], ref[16];
1573.     int qr, qc;
1574.     for (qr = 0; qr < 2; qr++)
1575.       for (qc = 0; qc < 2; qc++)
1576.         for (i = 0; i < 16; i++)
1577.           lc[qr][qc][i] = get4();
1578.     for (i = 0; i < 16; i++) {
1579.       int v = 0;
1580.       for (qr = 0; qr < 2; qr++)
1581.         for (qc = 0; qc < 2; qc++)
1582.           v += lc[qr][qc][i];
1583.       ref[i] = (v + 2) >> 2;
1584.     }
1585.     for (qr = 0; qr < 2; qr++) {
1586.       for (qc = 0; qc < 2; qc++) {
1587.         int cx[19], cf[19];
1588.         for (i = 0; i < 16; i++) {
1589.           cx[1+i] = lc[qr][qc][i];
1590.           cf[1+i] = ref[i];
1591.         }
1592.         cx[0] = cf[0] = 0;
1593.         cx[17] = cf[17] = ((unsigned) ref[15] * 65535) / lc[qr][qc][15];
1594.         cx[18] = cf[18] = 65535;
1595.         cubic_spline(cx, cf, 19);
1596.         for (row = (qr ? ph1.split_row : 0);
1597.              row < (qr ? raw_height : ph1.split_row); row++)
1598.           for (col = (qc ? ph1.split_col : 0);
1599.                col < (qc ? raw_width : ph1.split_col); col++)
1600.             RAW(row,col) = curve[RAW(row,col)];
1601.       }
1602.     }
1603.     qlin_applied = 1;
1604.   } else if (tag == 0x41e && !qmult_applied) { /* Quadrant multipliers */
1605.     float qmult[2][2] = { { 1, 1 }, { 1, 1 } };
1606.     get4(); get4(); get4(); get4();
1607.     qmult[0][0] = 1.0 + getreal(11);
1608.     get4(); get4(); get4(); get4(); get4();
1609.     qmult[0][1] = 1.0 + getreal(11);
1610.     get4(); get4(); get4();
1611.     qmult[1][0] = 1.0 + getreal(11);
1612.     get4(); get4(); get4();
1613.     qmult[1][1] = 1.0 + getreal(11);
1614.     for (row=0; row < raw_height; row++)
1615.       for (col=0; col < raw_width; col++) {
1616.         i = qmult[row >= ph1.split_row][col >= ph1.split_col] * RAW(row,col);
1617.         RAW(row,col) = LIM(i,0,65535);
1618.       }
1619.     qmult_applied = 1;
1620.   } else if (tag == 0x431 && !qmult_applied) { /* Quadrant combined */
```

```
1621.       ushort lc[2][2][7], ref[7];
1622.       int qr, qc;
1623.       for (i = 0; i < 7; i++)
1624.         ref[i] = get4();
1625.       for (qr = 0; qr < 2; qr++)
1626.         for (qc = 0; qc < 2; qc++)
1627.           for (i = 0; i < 7; i++)
1628.             lc[qr][qc][i] = get4();
1629.       for (qr = 0; qr < 2; qr++) {
1630.         for (qc = 0; qc < 2; qc++) {
1631.           int cx[9], cf[9];
1632.           for (i = 0; i < 7; i++) {
1633.             cx[1+i] = ref[i];
1634.             cf[1+i] = ((unsigned) ref[i] * lc[qr][qc][i]) / 10000;
1635.           }
1636.           cx[0] = cf[0] = 0;
1637.           cx[8] = cf[8] = 65535;
1638.           cubic_spline(cx, cf, 9);
1639.           for (row = (qr ? ph1.split_row : 0);
1640.                row < (qr ? raw_height : ph1.split_row); row++)
1641.             for (col = (qc ? ph1.split_col : 0);
1642.                  col < (qc ? raw_width : ph1.split_col); col++)
1643.               RAW(row,col) = curve[RAW(row,col)];
1644.         }
1645.       }
1646.       qmult_applied = 1;
1647.       qlin_applied = 1;
1648.     }
1649.     fseek (ifp, save, SEEK_SET);
1650.   }
1651.   if (off_412) {
1652.     fseek (ifp, off_412, SEEK_SET);
1653.     for (i=0; i < 9; i++) head[i] = get4() & 0x7fff;
1654.     yval[0] = (float *) calloc (head[1]*head[3] + head[2]*head[4], 6);
1655.     merror (yval[0], "phase_one_correct()");
1656.     yval[1] = (float  *) (yval[0] + head[1]*head[3]);
1657.     xval[0] = (ushort *) (yval[1] + head[2]*head[4]);
1658.     xval[1] = (ushort *) (xval[0] + head[1]*head[3]);
1659.     get2();
1660.     for (i=0; i < 2; i++)
1661.       for (j=0; j < head[i+1]*head[i+3]; j++)
1662.         yval[i][j] = getreal(11);
1663.     for (i=0; i < 2; i++)
1664.       for (j=0; j < head[i+1]*head[i+3]; j++)
1665.         xval[i][j] = get2();
1666.     for (row=0; row < raw_height; row++)
1667.       for (col=0; col < raw_width; col++) {
1668.         cfrac = (float) col * head[3] / raw_width;
1669.         cfrac -= cip = cfrac;
1670.         num = RAW(row,col) * 0.5;
1671.         for (i=cip; i < cip+2; i++) {
1672.           for (k=j=0; j < head[1]; j++)
1673.             if (num < xval[0][k = head[1]*i+j]) break;
1674.           frac = (j == 0 || j == head[1]) ? 0 :
1675.                  (xval[0][k] - num) / (xval[0][k] - xval[0][k-1]);
1676.           mult[i-cip] = yval[0][k-1] * frac + yval[0][k] * (1-frac);
1677.         }
1678.         i = ((mult[0] * (1-cfrac) + mult[1] * cfrac) * row + num) * 2;
1679.         RAW(row,col) = LIM(i,0,65535);
1680.       }
1681.     free (yval[0]);
1682.   }
1683. }
1684.
1685. void CLASS phase_one_load_raw()
```

```
{
  int a, b, i;
  ushort akey, bkey, mask;

  fseek (ifp, ph1.key_off, SEEK_SET);
  akey = get2();
  bkey = get2();
  mask = ph1.format == 1 ? 0x5555:0x1354;
  fseek (ifp, data_offset, SEEK_SET);
  read_shorts (raw_image, raw_width*raw_height);
  if (ph1.format)
    for (i=0; i < raw_width*raw_height; i+=2) {
      a = raw_image[i+0] ^ akey;
      b = raw_image[i+1] ^ bkey;
      raw_image[i+0] = (a & mask) | (b & ~mask);
      raw_image[i+1] = (b & mask) | (a & ~mask);
    }
}

unsigned CLASS ph1_bithuff (int nbits, ushort *huff)
{
  static UINT64 bitbuf=0;
  static int vbits=0;
  unsigned c;

  if (nbits == -1)
    return bitbuf = vbits = 0;
  if (nbits == 0) return 0;
  if (vbits < nbits) {
    bitbuf = bitbuf << 32 | get4();
    vbits += 32;
  }
  c = bitbuf << (64-vbits) >> (64-nbits);
  if (huff) {
    vbits -= huff[c] >> 8;
    return (uchar) huff[c];
  }
  vbits -= nbits;
  return c;
}
#define ph1_bits(n) ph1_bithuff(n,0)
#define ph1_huff(h) ph1_bithuff(*h,h+1)

void CLASS phase_one_load_raw_c()
{
  static const int length[] = { 8,7,6,9,11,10,5,12,14,13 };
  int *offset, len[2], pred[2], row, col, i, j;
  ushort *pixel;
  short (*cblack)[2], (*rblack)[2];

  pixel = (ushort *) calloc (raw_width*3 + raw_height*4, 2);
  merror (pixel, "phase_one_load_raw_c()");
  offset = (int *) (pixel + raw_width);
  fseek (ifp, strip_offset, SEEK_SET);
  for (row=0; row < raw_height; row++)
    offset[row] = get4();
  cblack = (short (*)[2]) (offset + raw_height);
  fseek (ifp, ph1.black_col, SEEK_SET);
  if (ph1.black_col)
    read_shorts ((ushort *) cblack[0], raw_height*2);
  rblack = cblack + raw_height;
  fseek (ifp, ph1.black_row, SEEK_SET);
  if (ph1.black_row)
    read_shorts ((ushort *) rblack[0], raw_width*2);
  for (i=0; i < 256; i++)
```

```
    curve[i] = i*i / 3.969 + 0.5;
  for (row=0; row < raw_height; row++) {
    fseek (ifp, data_offset + offset[row], SEEK_SET);
    ph1_bits(-1);
    pred[0] = pred[1] = 0;
    for (col=0; col < raw_width; col++) {
      if (col >= (raw_width & -8))
        len[0] = len[1] = 14;
      else if ((col & 7) == 0)
        for (i=0; i < 2; i++) {
          for (j=0; j < 5 && !ph1_bits(1); j++);
          if (j--) len[i] = length[j*2 + ph1_bits(1)];
        }
      if ((i = len[col & 1]) == 14)
        pixel[col] = pred[col & 1] = ph1_bits(16);
      else
        pixel[col] = pred[col & 1] += ph1_bits(i) + 1 - (1 << (i - 1));
      if (pred[col & 1] >> 16) derror();
      if (ph1.format == 5 && pixel[col] < 256)
        pixel[col] = curve[pixel[col]];
    }
    for (col=0; col < raw_width; col++) {
      i = (pixel[col] << 2*(ph1.format != 8)) - ph1.black
        + cblack[row][col >= ph1.split_col]
        + rblack[col][row >= ph1.split_row];
      if (i > 0) RAW(row,col) = i;
    }
  }
  free (pixel);
  maximum = 0xfffc - ph1.black;
}

void CLASS hasselblad_load_raw()
{
  struct jhead jh;
  int shot, row, col, *back[5], len[2], diff[12], pred, sh, f, s, c;
  unsigned upix, urow, ucol;
  ushort *ip;

  if (!ljpeg_start (&jh, 0)) return;
  order = 0x4949;
  ph1_bits(-1);
  back[4] = (int *) calloc (raw_width, 3*sizeof **back);
  merror (back[4], "hasselblad_load_raw()");
  FORC3 back[c] = back[4] + c*raw_width;
  cblack[6] >>= sh = tiff_samples > 1;
  shot = LIM(shot_select, 1, tiff_samples) - 1;
  for (row=0; row < raw_height; row++) {
    FORC4 back[(c+3) & 3] = back[c];
    for (col=0; col < raw_width; col+=2) {
      for (s=0; s < tiff_samples*2; s+=2) {
        FORC(2) len[c] = ph1_huff(jh.huff[0]);
        FORC(2) {
          diff[s+c] = ph1_bits(len[c]);
          if ((diff[s+c] & (1 << (len[c]-1))) == 0)
            diff[s+c] -= (1 << len[c]) - 1;
          if (diff[s+c] == 65535) diff[s+c] = -32768;
        }
      }
      for (s=col; s < col+2; s++) {
        pred = 0x8000 + load_flags;
        if (col) pred = back[2][s-2];
        if (col && row > 1) switch (jh.psv) {
          case 11: pred += back[0][s]/2 - back[0][s-2]/2;  break;
        }
```

```
1816.         f = (row & 1)*3 ^ ((col+s) & 1);
1817.         FORC (tiff_samples) {
1818.           pred += diff[(s & 1)*tiff_samples+c];
1819.           upix = pred >> sh & 0xffff;
1820.           if (raw_image && c == shot)
1821.             RAW(row,s) = upix;
1822.           if (image) {
1823.             urow = row-top_margin  + (c & 1);
1824.             ucol = col-left_margin - ((c >> 1) & 1);
1825.             ip = &image[urow*width+ucol][f];
1826.             if (urow < height && ucol < width)
1827.               *ip = c < 4 ? upix : (*ip + upix) >> 1;
1828.           }
1829.         }
1830.         back[2][s] = pred;
1831.       }
1832.     }
1833.   }
1834.   free (back[4]);
1835.   ljpeg_end (&jh);
1836.   if (image) mix_green = 1;
1837. }
1838.
1839. void CLASS leaf_hdr_load_raw()
1840. {
1841.   ushort *pixel=0;
1842.   unsigned tile=0, r, c, row, col;
1843.
1844.   if (!filters) {
1845.     pixel = (ushort *) calloc (raw_width, sizeof *pixel);
1846.     merror (pixel, "leaf_hdr_load_raw()");
1847.   }
1848.   FORC(tiff_samples)
1849.     for (r=0; r < raw_height; r++) {
1850.       if (r % tile_length == 0) {
1851.         fseek (ifp, data_offset + 4*tile++, SEEK_SET);
1852.         fseek (ifp, get4(), SEEK_SET);
1853.       }
1854.       if (filters && c != shot_select) continue;
1855.       if (filters) pixel = raw_image + r*raw_width;
1856.       read_shorts (pixel, raw_width);
1857.       if (!filters && (row = r - top_margin) < height)
1858.         for (col=0; col < width; col++)
1859.           image[row*width+col][c] = pixel[col+left_margin];
1860.     }
1861.   if (!filters) {
1862.     maximum = 0xffff;
1863.     raw_color = 1;
1864.     free (pixel);
1865.   }
1866. }
1867.
1868. void CLASS unpacked_load_raw()
1869. {
1870.   int row, col, bits=0;
1871.
1872.   while (1 << ++bits < maximum);
1873.   read_shorts (raw_image, raw_width*raw_height);
1874.   for (row=0; row < raw_height; row++)
1875.     for (col=0; col < raw_width; col++)
1876.       if ((RAW(row,col) >>= load_flags) >> bits
1877.         && (unsigned) (row-top_margin) < height
1878.         && (unsigned) (col-left_margin) < width) derror();
1879. }
1880.
```

```
1881.void CLASS sinar_4shot_load_raw()
1882.{
1883.  ushort *pixel;
1884.  unsigned shot, row, col, r, c;
1885.
1886.  if (raw_image) {
1887.    shot = LIM (shot_select, 1, 4) - 1;
1888.    fseek (ifp, data_offset + shot*4, SEEK_SET);
1889.    fseek (ifp, get4(), SEEK_SET);
1890.    unpacked_load_raw();
1891.    return;
1892.  }
1893.  pixel = (ushort *) calloc (raw_width, sizeof *pixel);
1894.  merror (pixel, "sinar_4shot_load_raw()");
1895.  for (shot=0; shot < 4; shot++) {
1896.    fseek (ifp, data_offset + shot*4, SEEK_SET);
1897.    fseek (ifp, get4(), SEEK_SET);
1898.    for (row=0; row < raw_height; row++) {
1899.      read_shorts (pixel, raw_width);
1900.      if ((r = row-top_margin - (shot >> 1 & 1)) >= height) continue;
1901.      for (col=0; col < raw_width; col++) {
1902.        if ((c = col-left_margin - (shot & 1)) >= width) continue;
1903.        image[r*width+c][(row & 1)*3 ^ (~col & 1)] = pixel[col];
1904.      }
1905.    }
1906.  }
1907.  free (pixel);
1908.  mix_green = 1;
1909.}
1910.
1911.void CLASS imacon_full_load_raw()
1912.{
1913.  int row, col;
1914.
1915.  if (!image) return;
1916.  for (row=0; row < height; row++)
1917.    for (col=0; col < width; col++)
1918.      read_shorts (image[row*width+col], 3);
1919.}
1920.
1921.void CLASS packed_load_raw()
1922.{
1923.  int vbits=0, bwide, rbits, bite, half, irow, row, col, val, i;
1924.  UINT64 bitbuf=0;
1925.
1926.  bwide = raw_width * tiff_bps / 8;
1927.  bwide += bwide & load_flags >> 9;
1928.  rbits = bwide * 8 - raw_width * tiff_bps;
1929.  if (load_flags & 1) bwide = bwide * 16 / 15;
1930.  bite = 8 + (load_flags & 56);
1931.  half = (raw_height+1) >> 1;
1932.  for (irow=0; irow < raw_height; irow++) {
1933.    row = irow;
1934.    if (load_flags & 2 &&
1935.        (row = irow % half * 2 + irow / half) == 1 &&
1936.        load_flags & 4) {
1937.      if (vbits=0, tiff_compress)
1938.        fseek (ifp, data_offset - (-half*bwide & -2048), SEEK_SET);
1939.      else {
1940.        fseek (ifp, 0, SEEK_END);
1941.        fseek (ifp, ftell(ifp) >> 3 << 2, SEEK_SET);
1942.      }
1943.    }
1944.    for (col=0; col < raw_width; col++) {
1945.      for (vbits -= tiff_bps; vbits < 0; vbits += bite) {
```

```
1946.         bitbuf <<= bite;
1947.         for (i=0; i < bite; i+=8)
1948.           bitbuf |= ((UINT64) fgetc(ifp) << i);
1949.       }
1950.       val = bitbuf << (64-tiff_bps-vbits) >> (64-tiff_bps);
1951.       RAW(row,col ^ (load_flags >> 6 & 3)) = val;
1952.       if (load_flags & 1 && (col % 10) == 9 && fgetc(ifp) &&
1953.         row < height+top_margin && col < width+left_margin) derror();
1954.     }
1955.     vbits -= rbits;
1956.   }
1957. }
1958.
1959. void CLASS nokia_load_raw()
1960. {
1961.   uchar  *data,  *dp;
1962.   int rev, dwide, row, col, c;
1963.   double sum[]={0,0};
1964.
1965.   rev = 3 * (order == 0x4949);
1966.   dwide = (raw_width * 5 + 1) / 4;
1967.   data = (uchar *) malloc (dwide*2);
1968.   merror (data, "nokia_load_raw()");
1969.   for (row=0; row < raw_height; row++) {
1970.     if (fread (data+dwide, 1, dwide, ifp) < dwide) derror();
1971.     FORC(dwide) data[c] = data[dwide+(c ^ rev)];
1972.     for (dp=data, col=0; col < raw_width; dp+=5, col+=4)
1973.       FORC4 RAW(row,col+c) = (dp[c] << 2) | (dp[4] >> (c << 1) & 3);
1974.   }
1975.   free (data);
1976.   maximum = 0x3ff;
1977.   if (strcmp(make,"OmniVision")) return;
1978.   row = raw_height/2;
1979.   FORC(width-1) {
1980.     sum[ c & 1] += SQR(RAW(row,c)-RAW(row+1,c+1));
1981.     sum[~c & 1] += SQR(RAW(row+1,c)-RAW(row,c+1));
1982.   }
1983.   if (sum[1] > sum[0]) filters = 0x4b4b4b4b;
1984. }
1985.
1986. void CLASS canon_rmf_load_raw()
1987. {
1988.   int row, col, bits, orow, ocol, c;
1989.
1990.   for (row=0; row < raw_height; row++)
1991.     for (col=0; col < raw_width-2; col+=3) {
1992.       bits = get4();
1993.       FORC3 {
1994.         orow = row;
1995.         if ((ocol = col+c-4) < 0) {
1996.           ocol += raw_width;
1997.           if ((orow -= 2) < 0)
1998.             orow += raw_height;
1999.         }
2000.         RAW(orow,ocol) = curve[bits >> (10*c+2) & 0x3ff];
2001.       }
2002.     }
2003.   maximum = curve[0x3ff];
2004. }
2005.
2006. unsigned CLASS pana_bits (int nbits)
2007. {
2008.   static uchar buf[0x4000];
2009.   static int vbits;
2010.   int byte;
```

```
2011.
2012.  if (!nbits) return vbits=0;
2013.  if (!vbits) {
2014.    fread (buf+load_flags, 1, 0x4000-load_flags, ifp);
2015.    fread (buf, 1, load_flags, ifp);
2016.  }
2017.  vbits = (vbits - nbits) & 0x1ffff;
2018.  byte = vbits >> 3 ^ 0x3ff0;
2019.  return (buf[byte] | buf[byte+1] << 8) >> (vbits & 7) & ~(-1 << nbits);
2020.}
2021.
2022.void CLASS panasonic_load_raw()
2023.{
2024.  int row, col, i, j, sh=0, pred[2], nonz[2];
2025.
2026.  pana_bits(0);
2027.  for (row=0; row < height; row++)
2028.    for (col=0; col < raw_width; col++) {
2029.      if ((i = col % 14) == 0)
2030.        pred[0] = pred[1] = nonz[0] = nonz[1] = 0;
2031.      if (i % 3 == 2) sh = 4 >> (3 - pana_bits(2));
2032.      if (nonz[i & 1]) {
2033.        if ((j = pana_bits(8))) {
2034.          if ((pred[i & 1] -= 0x80 << sh) < 0 || sh == 4)
2035.               pred[i & 1] &= ~(-1 << sh);
2036.          pred[i & 1] += j << sh;
2037.        }
2038.      } else if ((nonz[i & 1] = pana_bits(8)) || i > 11)
2039.        pred[i & 1] = nonz[i & 1] << 4 | pana_bits(4);
2040.      if ((RAW(row,col) = pred[col & 1]) > 4098 && col < width) derror();
2041.    }
2042.}
2043.
2044.void CLASS olympus_load_raw()
2045.{
2046.  ushort huff[4096];
2047.  int row, col, nbits, sign, low, high, i, c, w, n, nw;
2048.  int acarry[2][3], *carry, pred, diff;
2049.
2050.  huff[n=0] = 0xc0c;
2051.  for (i=12; i--; )
2052.    FORC(2048 >> i) huff[++n] = (i+1) << 8 | i;
2053.  fseek (ifp, 7, SEEK_CUR);
2054.  getbits(-1);
2055.  for (row=0; row < height; row++) {
2056.    memset (acarry, 0, sizeof acarry);
2057.    for (col=0; col < raw_width; col++) {
2058.      carry = acarry[col & 1];
2059.      i = 2 * (carry[2] < 3);
2060.      for (nbits=2+i; (ushort) carry[0] >> (nbits+i); nbits++);
2061.      low = (sign = getbits(3)) & 3;
2062.      sign = sign << 29 >> 31;
2063.      if ((high = getbithuff(12,huff)) == 12)
2064.        high = getbits(16-nbits) >> 1;
2065.      carry[0] = (high << nbits) | getbits(nbits);
2066.      diff = (carry[0] ^ sign) + carry[1];
2067.      carry[1] = (diff*3 + carry[1]) >> 5;
2068.      carry[2] = carry[0] > 16 ? 0 : carry[2]+1;
2069.      if (col >= width) continue;
2070.      if (row < 2 && col < 2) pred = 0;
2071.      else if (row < 2) pred = RAW(row,col-2);
2072.      else if (col < 2) pred = RAW(row-2,col);
2073.      else {
2074.        w  = RAW(row,col-2);
2075.        n  = RAW(row-2,col);
```

```
2076.         nw = RAW(row-2,col-2);
2077.         if ((w < nw && nw < n) || (n < nw && nw < w)) {
2078.           if (ABS(w-nw) > 32 || ABS(n-nw) > 32)
2079.             pred = w + n - nw;
2080.           else pred = (w + n) >> 1;
2081.         } else pred = ABS(w-nw) > ABS(n-nw) ? w : n;
2082.       }
2083.       if ((RAW(row,col) = pred + ((diff << 2) | low)) >> 12) derror();
2084.     }
2085.   }
2086. }
2087.
2088. void CLASS canon_crx_load_raw()
2089. {
2090. }
2091.
2092. void CLASS fuji_xtrans_load_raw()
2093. {
2094. }
2095.
2096. void CLASS minolta_rd175_load_raw()
2097. {
2098.   uchar pixel[768];
2099.   unsigned irow, box, row, col;
2100.
2101.   for (irow=0; irow < 1481; irow++) {
2102.     if (fread (pixel, 1, 768, ifp) < 768) derror();
2103.     box = irow / 82;
2104.     row = irow % 82 * 12 + ((box < 12) ? box | 1 : (box-12)*2);
2105.     switch (irow) {
2106.       case 1477: case 1479: continue;
2107.       case 1476: row = 984; break;
2108.       case 1480: row = 985; break;
2109.       case 1478: row = 985; box = 1;
2110.     }
2111.     if ((box < 12) && (box & 1)) {
2112.       for (col=0; col < 1533; col++, row ^= 1)
2113.         if (col != 1) RAW(row,col) = (col+1) & 2 ?
2114.                       pixel[col/2-1] + pixel[col/2+1] : pixel[col/2] << 1;
2115.       RAW(row,1)    = pixel[1]   << 1;
2116.       RAW(row,1533) = pixel[765] << 1;
2117.     } else
2118.       for (col=row & 1; col < 1534; col+=2)
2119.         RAW(row,col) = pixel[col/2] << 1;
2120.   }
2121.   maximum = 0xff << 1;
2122. }
2123.
2124. void CLASS quicktake_100_load_raw()
2125. {
2126.   uchar pixel[484][644];
2127.   static const short gstep[16] =
2128.   { -89,-60,-44,-32,-22,-15,-8,-2,2,8,15,22,32,44,60,89 };
2129.   static const short rstep[6][4] =
2130.   { {  -3,-1,1,3  }, {  -5,-1,1,5  }, {  -8,-2,2,8  },
2131.     { -13,-3,3,13 }, { -19,-4,4,19 }, { -28,-6,6,28 } };
2132.   static const short curve[256] =
2133.   { 0,1,2,3,4,5,6,7,8,9,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,
2134.     28,29,30,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,53,
2135.     54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,74,75,76,77,78,
2136.     79,80,81,82,83,84,86,88,90,92,94,97,99,101,103,105,107,110,112,114,116,
2137.     118,120,123,125,127,129,131,134,136,138,140,142,144,147,149,151,153,155,
2138.     158,160,162,164,166,168,171,173,175,177,179,181,184,186,188,190,192,195,
2139.     197,199,201,203,205,208,210,212,214,216,218,221,223,226,230,235,239,244,
2140.     248,252,257,261,265,270,274,278,283,287,291,296,300,305,309,313,318,322,
```

```
    326,331,335,339,344,348,352,357,361,365,370,374,379,383,387,392,396,400,
    405,409,413,418,422,426,431,435,440,444,448,453,457,461,466,470,474,479,
    483,487,492,496,500,508,519,531,542,553,564,575,587,598,609,620,631,643,
    654,665,676,687,698,710,721,732,743,754,766,777,788,799,810,822,833,844,
    855,866,878,889,900,911,922,933,945,956,967,978,989,1001,1012,1023 };
  int rb, row, col, sharp, val=0;

  getbits(-1);
  memset (pixel, 0x80, sizeof pixel);
  for (row=2; row < height+2; row++) {
    for (col=2+(row & 1); col < width+2; col+=2) {
      val = ((pixel[row-1][col-1] + 2*pixel[row-1][col+1] +
                pixel[row][col-2]) >> 2) + gstep[getbits(4)];
      pixel[row][col] = val = LIM(val,0,255);
      if (col < 4)
        pixel[row][col-2] = pixel[row+1][~row & 1] = val;
      if (row == 2)
        pixel[row-1][col+1] = pixel[row-1][col+3] = val;
    }
    pixel[row][col] = val;
  }
  for (rb=0; rb < 2; rb++)
    for (row=2+rb; row < height+2; row+=2)
      for (col=3-(row & 1); col < width+2; col+=2) {
        if (row < 4 || col < 4) sharp = 2;
        else {
          val = ABS(pixel[row-2][col] - pixel[row][col-2])
              + ABS(pixel[row-2][col] - pixel[row-2][col-2])
              + ABS(pixel[row][col-2] - pixel[row-2][col-2]);
          sharp = val <  4 ? 0 : val <  8 ? 1 : val < 16 ? 2 :
                  val < 32 ? 3 : val < 48 ? 4 : 5;
        }
        val = ((pixel[row-2][col] + pixel[row][col-2]) >> 1)
              + rstep[sharp][getbits(2)];
        pixel[row][col] = val = LIM(val,0,255);
        if (row < 4) pixel[row-2][col+2] = val;
        if (col < 4) pixel[row+2][col-2] = val;
      }
  for (row=2; row < height+2; row++)
    for (col=3-(row & 1); col < width+2; col+=2) {
      val = ((pixel[row][col-1] + (pixel[row][col] << 2) +
              pixel[row][col+1]) >> 1) - 0x100;
      pixel[row][col] = LIM(val,0,255);
    }
  for (row=0; row < height; row++)
    for (col=0; col < width; col++)
      RAW(row,col) = curve[pixel[row+2][col+2]];
  maximum = 0x3ff;
}

#define radc_token(tree) ((signed char) getbithuff(8,huff[tree]))

#define FORYX for (y=1; y < 3; y++) for (x=col+1; x >= col; x--)

#define PREDICTOR (c ? (buf[c][y-1][x] + buf[c][y][x+1]) / 2 \
: (buf[c][y-1][x+1] + 2*buf[c][y-1][x] + buf[c][y][x+1]) / 4)

void CLASS kodak_radc_load_raw()
{
  static const char src[] = {
    1,1, 2,3, 3,4, 4,2, 5,7, 6,5, 7,6, 7,8,
    1,0, 2,1, 3,3, 4,4, 5,2, 6,7, 7,6, 8,5, 8,8,
    2,1, 2,3, 3,0, 3,2, 3,4, 4,6, 5,5, 6,7, 6,8,
    2,0, 2,1, 2,3, 3,2, 4,4, 5,6, 6,7, 7,5, 7,8,
    2,1, 2,4, 3,0, 3,2, 3,3, 4,7, 5,5, 6,6, 6,8,
```

```
    2,3, 3,1, 3,2, 3,4, 3,5, 3,6, 4,7, 5,0, 5,8,
    2,3, 2,6, 3,0, 3,1, 4,4, 4,5, 4,7, 5,2, 5,8,
    2,4, 2,7, 3,3, 3,6, 4,1, 4,2, 4,5, 5,0, 5,8,
    2,6, 3,1, 3,3, 3,5, 3,7, 3,8, 4,0, 5,2, 5,4,
    2,0, 2,1, 3,2, 3,3, 4,4, 4,5, 5,6, 5,7, 4,8,
    1,0, 2,2, 2,-2,
    1,-3, 1,3,
    2,-17, 2,-5, 2,5, 2,17,
    2,-7, 2,2, 2,9, 2,18,
    2,-18, 2,-9, 2,-2, 2,7,
    2,-28, 2,28, 3,-49, 3,-9, 3,9, 4,49, 5,-79, 5,79,
    2,-1, 2,13, 2,26, 3,39, 4,-16, 5,55, 6,-37, 6,76,
    2,-26, 2,-13, 2,1, 3,-39, 4,16, 5,-55, 6,-76, 6,37
  };
  ushort huff[19][256];
  int row, col, tree, nreps, rep, step, i, c, s, r, x, y, val;
  short last[3] = { 16,16,16 }, mul[3], buf[3][3][386];
  static const ushort pt[] =
    { 0,0, 1280,1344, 2320,3616, 3328,8000, 4095,16383, 65535,16383 };

  for (i=2; i < 12; i+=2)
    for (c=pt[i-2]; c <= pt[i]; c++)
      curve[c] = (float)
        (c-pt[i-2]) / (pt[i]-pt[i-2]) * (pt[i+1]-pt[i-1]) + pt[i-1] + 0.5;
  for (s=i=0; i < sizeof src; i+=2)
    FORC(256 >> src[i])
      ((ushort *)huff)[s++] = src[i] << 8 | (uchar) src[i+1];
  s = kodak_cbpp == 243 ? 2 : 3;
  FORC(256) huff[18][c] = (8-s) << 8 | c >> s << s | 1 << (s-1);
  getbits(-1);
  for (i=0; i < sizeof(buf)/sizeof(short); i++)
    ((short *)buf)[i] = 2048;
  for (row=0; row < height; row+=4) {
    FORC3 mul[c] = getbits(6);
    FORC3 {
      val = ((0x1000000/last[c] + 0x7ff) >> 12) * mul[c];
      s = val > 65564 ? 10:12;
      x = ~(-1 << (s-1));
      val <<= 12-s;
      for (i=0; i < sizeof(buf[0])/sizeof(short); i++)
        ((short *)buf[c])[i] = (((short *)buf[c])[i] * val + x) >> s;
      last[c] = mul[c];
      for (r=0; r <= !c; r++) {
        buf[c][1][width/2] = buf[c][2][width/2] = mul[c] << 7;
        for (tree=1, col=width/2; col > 0; ) {
          if ((tree = radc_token(tree))) {
            col -= 2;
            if (tree == 8)
              FORYX buf[c][y][x] = (uchar) radc_token(18) * mul[c];
            else
              FORYX buf[c][y][x] = radc_token(tree+10) * 16 + PREDICTOR;
          } else
            do {
              nreps = (col > 2) ? radc_token(9) + 1 : 1;
              for (rep=0; rep < 8 && rep < nreps && col > 0; rep++) {
                col -= 2;
                FORYX buf[c][y][x] = PREDICTOR;
                if (rep & 1) {
                  step = radc_token(10) << 4;
                  FORYX buf[c][y][x] += step;
                }
              }
            } while (nreps == 9);
        }
        for (y=0; y < 2; y++)
```

```
2271.           for (x=0; x < width/2; x++) {
2272.             val = (buf[c][y+1][x] << 4) / mul[c];
2273.             if (val < 0) val = 0;
2274.             if (c) RAW(row+y*2+c-1,x*2+2-c) = val;
2275.             else   RAW(row+r*2+y,x*2+y) = val;
2276.           }
2277.         memcpy (buf[c][0]+!c, buf[c][2], sizeof buf[c][0]-2*!c);
2278.       }
2279.     }
2280.     for (y=row; y < row+4; y++)
2281.       for (x=0; x < width; x++)
2282.         if ((x+y) & 1) {
2283.           r = x ? x-1 : x+1;
2284.           s = x+1 < width ? x+1 : x-1;
2285.           val = (RAW(y,x)-2048)*2 + (RAW(y,r)+RAW(y,s))/2;
2286.           if (val < 0) val = 0;
2287.           RAW(y,x) = val;
2288.         }
2289.   }
2290.   for (i=0; i < height*width; i++)
2291.     raw_image[i] = curve[raw_image[i]];
2292.   maximum = 0x3fff;
2293. }
2294.
2295. #undef FORYX
2296. #undef PREDICTOR
2297.
2298. #ifdef NO_JPEG
2299. void CLASS kodak_jpeg_load_raw() {}
2300. void CLASS lossy_dng_load_raw() {}
2301. #else
2302.
2303. METHODDEF(boolean)
2304. fill_input_buffer (j_decompress_ptr cinfo)
2305. {
2306.   static uchar jpeg_buffer[4096];
2307.   size_t nbytes;
2308.
2309.   nbytes = fread (jpeg_buffer, 1, 4096, ifp);
2310.   swab (jpeg_buffer, jpeg_buffer, nbytes);
2311.   cinfo->src->next_input_byte = jpeg_buffer;
2312.   cinfo->src->bytes_in_buffer = nbytes;
2313.   return TRUE;
2314. }
2315.
2316. void CLASS kodak_jpeg_load_raw()
2317. {
2318.   struct jpeg_decompress_struct cinfo;
2319.   struct jpeg_error_mgr jerr;
2320.   JSAMPARRAY buf;
2321.   JSAMPLE (*pixel)[3];
2322.   int row, col;
2323.
2324.   cinfo.err = jpeg_std_error (&jerr);
2325.   jpeg_create_decompress (&cinfo);
2326.   jpeg_stdio_src (&cinfo, ifp);
2327.   cinfo.src->fill_input_buffer = fill_input_buffer;
2328.   jpeg_read_header (&cinfo, TRUE);
2329.   jpeg_start_decompress (&cinfo);
2330.   if ((cinfo.output_width      != width  ) ||
2331.       (cinfo.output_height*2   != height ) ||
2332.       (cinfo.output_components != 3      )) {
2333.     fprintf (stderr,_("%s: incorrect JPEG dimensions\n"), ifname);
2334.     jpeg_destroy_decompress (&cinfo);
2335.     longjmp (failure, 3);
```

```
2336.  }
2337.  buf = (*cinfo.mem->alloc_sarray)
2338.                ((j_common_ptr) &cinfo, JPOOL_IMAGE, width*3, 1);
2339.
2340.  while (cinfo.output_scanline < cinfo.output_height) {
2341.    row = cinfo.output_scanline * 2;
2342.    jpeg_read_scanlines (&cinfo, buf, 1);
2343.    pixel = (JSAMPLE (*)[3]) buf[0];
2344.    for (col=0; col < width; col+=2) {
2345.      RAW(row+0,col+0) = pixel[col+0][1] << 1;
2346.      RAW(row+1,col+1) = pixel[col+1][1] << 1;
2347.      RAW(row+0,col+1) = pixel[col][0] + pixel[col+1][0];
2348.      RAW(row+1,col+0) = pixel[col][2] + pixel[col+1][2];
2349.    }
2350.  }
2351.  jpeg_finish_decompress (&cinfo);
2352.  jpeg_destroy_decompress (&cinfo);
2353.  maximum = 0xff << 1;
2354.}
2355.
2356.void CLASS gamma_curve (double pwr, double ts, int mode, int imax);
2357.
2358.void CLASS lossy_dng_load_raw()
2359.{
2360.  struct jpeg_decompress_struct cinfo;
2361.  struct jpeg_error_mgr jerr;
2362.  JSAMPARRAY buf;
2363.  JSAMPLE (*pixel)[3];
2364.  unsigned sorder=order, ntags, opcode, deg, i, j, c;
2365.  unsigned save=data_offset-4, trow=0, tcol=0, row, col;
2366.  ushort cur[3][256];
2367.  double coeff[9], tot;
2368.
2369.  if (meta_offset) {
2370.    fseek (ifp, meta_offset, SEEK_SET);
2371.    order = 0x4d4d;
2372.    ntags = get4();
2373.    while (ntags--) {
2374.      opcode = get4(); get4(); get4();
2375.      if (opcode != 8)
2376.      { fseek (ifp, get4(), SEEK_CUR); continue; }
2377.      fseek (ifp, 20, SEEK_CUR);
2378.      if ((c = get4()) > 2) break;
2379.      fseek (ifp, 12, SEEK_CUR);
2380.      if ((deg = get4()) > 8) break;
2381.      for (i=0; i <= deg && i < 9; i++)
2382.        coeff[i] = getreal(12);
2383.      for (i=0; i < 256; i++) {
2384.        for (tot=j=0; j <= deg; j++)
2385.          tot += coeff[j] * pow(i/255.0, j);
2386.        cur[c][i] = tot*0xffff;
2387.      }
2388.    }
2389.    order = sorder;
2390.  } else {
2391.    gamma_curve (1/2.4, 12.92, 1, 255);
2392.    FORC3 memcpy (cur[c], curve, sizeof cur[0]);
2393.  }
2394.  cinfo.err = jpeg_std_error (&jerr);
2395.  jpeg_create_decompress (&cinfo);
2396.  while (trow < raw_height) {
2397.    fseek (ifp, save+=4, SEEK_SET);
2398.    if (tile_length < INT_MAX)
2399.      fseek (ifp, get4(), SEEK_SET);
2400.    jpeg_stdio_src (&cinfo, ifp);
```

```
2401.    jpeg_read_header (&cinfo, TRUE);
2402.    jpeg_start_decompress (&cinfo);
2403.    buf = (*cinfo.mem->alloc_sarray)
2404.        ((j_common_ptr) &cinfo, JPOOL_IMAGE, cinfo.output_width*3, 1);
2405.    while (cinfo.output_scanline < cinfo.output_height &&
2406.        (row = trow + cinfo.output_scanline) < height) {
2407.      jpeg_read_scanlines (&cinfo, buf, 1);
2408.      pixel = (JSAMPLE (*)[3]) buf[0];
2409.      for (col=0; col < cinfo.output_width && tcol+col < width; col++) {
2410.        FORC3 image[row*width+tcol+col][c] = cur[c][pixel[col][c]];
2411.      }
2412.    }
2413.    jpeg_abort_decompress (&cinfo);
2414.    if ((tcol += tile_width) >= raw_width)
2415.      trow += tile_length + (tcol = 0);
2416.  }
2417.  jpeg_destroy_decompress (&cinfo);
2418.  maximum = 0xffff;
2419.}
2420.#endif
2421.
2422.void CLASS kodak_dc120_load_raw()
2423.{
2424.  static const int mul[4] = { 162, 192, 187,  92 };
2425.  static const int add[4] = {   0, 636, 424, 212 };
2426.  uchar pixel[848];
2427.  int row, shift, col;
2428.
2429.  for (row=0; row < height; row++) {
2430.    if (fread (pixel, 1, 848, ifp) < 848) derror();
2431.    shift = row * mul[row & 3] + add[row & 3];
2432.    for (col=0; col < width; col++)
2433.      RAW(row,col) = (ushort) pixel[(col + shift) % 848];
2434.  }
2435.  maximum = 0xff;
2436.}
2437.
2438.void CLASS eight_bit_load_raw()
2439.{
2440.  uchar *pixel;
2441.  unsigned row, col;
2442.
2443.  pixel = (uchar *) calloc (raw_width, sizeof *pixel);
2444.  merror (pixel, "eight_bit_load_raw()");
2445.  for (row=0; row < raw_height; row++) {
2446.    if (fread (pixel, 1, raw_width, ifp) < raw_width) derror();
2447.    for (col=0; col < raw_width; col++)
2448.      RAW(row,col) = curve[pixel[col]];
2449.  }
2450.  free (pixel);
2451.  maximum = curve[0xff];
2452.}
2453.
2454.void CLASS kodak_c330_load_raw()
2455.{
2456.  uchar *pixel;
2457.  int row, col, y, cb, cr, rgb[3], c;
2458.
2459.  pixel = (uchar *) calloc (raw_width, 2*sizeof *pixel);
2460.  merror (pixel, "kodak_c330_load_raw()");
2461.  for (row=0; row < height; row++) {
2462.    if (fread (pixel, raw_width, 2, ifp) < 2) derror();
2463.    if (load_flags && (row & 31) == 31)
2464.      fseek (ifp, raw_width*32, SEEK_CUR);
2465.    for (col=0; col < width; col++) {
```

```
2466.       y  = pixel[col*2];
2467.       cb = pixel[(col*2 & -4) | 1] - 128;
2468.       cr = pixel[(col*2 & -4) | 3] - 128;
2469.       rgb[1] = y - ((cb + cr + 2) >> 2);
2470.       rgb[2] = rgb[1] + cb;
2471.       rgb[0] = rgb[1] + cr;
2472.       FORC3 image[row*width+col][c] = curve[LIM(rgb[c],0,255)];
2473.     }
2474.   }
2475.   free (pixel);
2476.   maximum = curve[0xff];
2477. }
2478.
2479. void CLASS kodak_c603_load_raw()
2480. {
2481.   uchar *pixel;
2482.   int row, col, y, cb, cr, rgb[3], c;
2483.
2484.   pixel = (uchar *) calloc (raw_width, 3*sizeof *pixel);
2485.   merror (pixel, "kodak_c603_load_raw()");
2486.   for (row=0; row < height; row++) {
2487.     if (~row & 1)
2488.       if (fread (pixel, raw_width, 3, ifp) < 3) derror();
2489.     for (col=0; col < width; col++) {
2490.       y  = pixel[width*2*(row & 1) + col];
2491.       cb = pixel[width + (col & -2)]   - 128;
2492.       cr = pixel[width + (col & -2)+1] - 128;
2493.       rgb[1] = y - ((cb + cr + 2) >> 2);
2494.       rgb[2] = rgb[1] + cb;
2495.       rgb[0] = rgb[1] + cr;
2496.       FORC3 image[row*width+col][c] = curve[LIM(rgb[c],0,255)];
2497.     }
2498.   }
2499.   free (pixel);
2500.   maximum = curve[0xff];
2501. }
2502.
2503. void CLASS kodak_262_load_raw()
2504. {
2505.   static const uchar kodak_tree[2][26] =
2506.   { { 0,1,5,1,1,2,0,0,0,0,0,0,0,0,0,0, 0,1,2,3,4,5,6,7,8,9 },
2507.     { 0,3,1,1,1,1,1,2,0,0,0,0,0,0,0,0, 0,1,2,3,4,5,6,7,8,9 } };
2508.   ushort *huff[2];
2509.   uchar *pixel;
2510.   int *strip, ns, c, row, col, chess, pi=0, pi1, pi2, pred, val;
2511.
2512.   FORC(2) huff[c] = make_decoder (kodak_tree[c]);
2513.   ns = (raw_height+63) >> 5;
2514.   pixel = (uchar *) malloc (raw_width*32 + ns*4);
2515.   merror (pixel, "kodak_262_load_raw()");
2516.   strip = (int *) (pixel + raw_width*32);
2517.   order = 0x4d4d;
2518.   FORC(ns) strip[c] = get4();
2519.   for (row=0; row < raw_height; row++) {
2520.     if ((row & 31) == 0) {
2521.       fseek (ifp, strip[row >> 5], SEEK_SET);
2522.       getbits(-1);
2523.       pi = 0;
2524.     }
2525.     for (col=0; col < raw_width; col++) {
2526.       chess = (row + col) & 1;
2527.       pi1 = chess ? pi-2           : pi-raw_width-1;
2528.       pi2 = chess ? pi-2*raw_width : pi-raw_width+1;
2529.       if (col <= chess) pi1 = -1;
2530.       if (pi1 < 0) pi1 = pi2;
```

```
2531.      if (pi2 < 0) pi2 = pi1;
2532.      if (pi1 < 0 && col > 1) pi1 = pi2 = pi-2;
2533.      pred = (pi1 < 0) ? 0 : (pixel[pi1] + pixel[pi2]) >> 1;
2534.      pixel[pi] = val = pred + ljpeg_diff (huff[chess]);
2535.      if (val >> 8) derror();
2536.      val = curve[pixel[pi++]];
2537.      RAW(row,col) = val;
2538.    }
2539.  }
2540.  free (pixel);
2541.  FORC(2) free (huff[c]);
2542.}
2543.
2544.int CLASS kodak_65000_decode (short *out, int bsize)
2545.{
2546.  uchar c, blen[768];
2547.  ushort raw[6];
2548.  INT64 bitbuf=0;
2549.  int save, bits=0, i, j, len, diff;
2550.
2551.  save = ftell(ifp);
2552.  bsize = (bsize + 3) & -4;
2553.  for (i=0; i < bsize; i+=2) {
2554.    c = fgetc(ifp);
2555.    if ((blen[i  ] = c & 15) > 12 ||
2556.        (blen[i+1] = c >> 4) > 12 ) {
2557.      fseek (ifp, save, SEEK_SET);
2558.      for (i=0; i < bsize; i+=8) {
2559.        read_shorts (raw, 6);
2560.        out[i  ] = raw[0] >> 12 << 8 | raw[2] >> 12 << 4 | raw[4] >> 12;
2561.        out[i+1] = raw[1] >> 12 << 8 | raw[3] >> 12 << 4 | raw[5] >> 12;
2562.        for (j=0; j < 6; j++)
2563.          out[i+2+j] = raw[j] & 0xfff;
2564.      }
2565.      return 1;
2566.    }
2567.  }
2568.  if ((bsize & 7) == 4) {
2569.    bitbuf  = fgetc(ifp) << 8;
2570.    bitbuf += fgetc(ifp);
2571.    bits = 16;
2572.  }
2573.  for (i=0; i < bsize; i++) {
2574.    len = blen[i];
2575.    if (bits < len) {
2576.      for (j=0; j < 32; j+=8)
2577.        bitbuf += (INT64) fgetc(ifp) << (bits+(j^8));
2578.      bits += 32;
2579.    }
2580.    diff = bitbuf & (0xffff >> (16-len));
2581.    bitbuf >>= len;
2582.    bits -= len;
2583.    if ((diff & (1 << (len-1))) == 0)
2584.      diff -= (1 << len) - 1;
2585.    out[i] = diff;
2586.  }
2587.  return 0;
2588.}
2589.
2590.void CLASS kodak_65000_load_raw()
2591.{
2592.  short buf[256];
2593.  int row, col, len, pred[2], ret, i;
2594.
2595.  for (row=0; row < height; row++)
```

```
2596.    for (col=0; col < width; col+=256) {
2597.      pred[0] = pred[1] = 0;
2598.      len = MIN (256, width-col);
2599.      ret = kodak_65000_decode (buf, len);
2600.      for (i=0; i < len; i++)
2601.        if ((RAW(row,col+i) =   curve[ret ? buf[i] :
2602.                (pred[i & 1] += buf[i])]) >> 12) derror();
2603.    }
2604.}
2605.
2606.void CLASS kodak_ycbcr_load_raw()
2607.{
2608.  short buf[384], *bp;
2609.  int row, col, len, c, i, j, k, y[2][2], cb, cr, rgb[3];
2610.  ushort *ip;
2611.
2612.  if (!image) return;
2613.  for (row=0; row < height; row+=2)
2614.    for (col=0; col < width; col+=128) {
2615.      len = MIN (128, width-col);
2616.      kodak_65000_decode (buf, len*3);
2617.      y[0][1] = y[1][1] = cb = cr = 0;
2618.      for (bp=buf, i=0; i < len; i+=2, bp+=2) {
2619.        cb += bp[4];
2620.        cr += bp[5];
2621.        rgb[1] = -((cb + cr + 2) >> 2);
2622.        rgb[2] = rgb[1] + cb;
2623.        rgb[0] = rgb[1] + cr;
2624.        for (j=0; j < 2; j++)
2625.          for (k=0; k < 2; k++) {
2626.            if ((y[j][k] = y[j][k^1] + *bp++) >> 10) derror();
2627.            ip = image[(row+j)*width + col+i+k];
2628.            FORC3 ip[c] = curve[LIM(y[j][k]+rgb[c], 0, 0xfff)];
2629.          }
2630.      }
2631.    }
2632.}
2633.
2634.void CLASS kodak_rgb_load_raw()
2635.{
2636.  short buf[768], *bp;
2637.  int row, col, len, c, i, rgb[3];
2638.  ushort *ip=image[0];
2639.
2640.  for (row=0; row < height; row++)
2641.    for (col=0; col < width; col+=256) {
2642.      len = MIN (256, width-col);
2643.      kodak_65000_decode (buf, len*3);
2644.      memset (rgb, 0, sizeof rgb);
2645.      for (bp=buf, i=0; i < len; i++, ip+=4)
2646.        FORC3 if ((ip[c] = rgb[c] += *bp++) >> 12) derror();
2647.    }
2648.}
2649.
2650.void CLASS kodak_thumb_load_raw()
2651.{
2652.  int row, col;
2653.  colors = thumb_misc >> 5;
2654.  for (row=0; row < height; row++)
2655.    for (col=0; col < width; col++)
2656.      read_shorts (image[row*width+col], colors);
2657.  maximum = (1 << (thumb_misc & 31)) - 1;
2658.}
2659.
2660.void CLASS sony_decrypt (unsigned *data, int len, int start, int key)
```

```
2661.{
2662.  static unsigned pad[128], p;
2663.
2664.  if (start) {
2665.    for (p=0; p < 4; p++)
2666.      pad[p] = key = key * 48828125 + 1;
2667.    pad[3] = pad[3] << 1 | (pad[0]^pad[2]) >> 31;
2668.    for (p=4; p < 127; p++)
2669.      pad[p] = (pad[p-4]^pad[p-2]) << 1 | (pad[p-3]^pad[p-1]) >> 31;
2670.    for (p=0; p < 127; p++)
2671.      pad[p] = htonl(pad[p]);
2672.  }
2673.  while (len-- && p++)
2674.    *data++ ^= pad[(p-1) & 127] = pad[p & 127] ^ pad[(p+64) & 127];
2675.}
2676.
2677.void CLASS sony_load_raw()
2678.{
2679.  uchar head[40];
2680.  ushort *pixel;
2681.  unsigned i, key, row, col;
2682.
2683.  fseek (ifp, 200896, SEEK_SET);
2684.  fseek (ifp, (unsigned) fgetc(ifp)*4 - 1, SEEK_CUR);
2685.  order = 0x4d4d;
2686.  key = get4();
2687.  fseek (ifp, 164600, SEEK_SET);
2688.  fread (head, 1, 40, ifp);
2689.  sony_decrypt ((unsigned *) head, 10, 1, key);
2690.  for (i=26; i-- > 22; )
2691.    key = key << 8 | head[i];
2692.  fseek (ifp, data_offset, SEEK_SET);
2693.  for (row=0; row < raw_height; row++) {
2694.    pixel = raw_image + row*raw_width;
2695.    if (fread (pixel, 2, raw_width, ifp) < raw_width) derror();
2696.    sony_decrypt ((unsigned *) pixel, raw_width/2, !row, key);
2697.    for (col=0; col < raw_width; col++)
2698.      if ((pixel[col] = ntohs(pixel[col])) >> 14) derror();
2699.  }
2700.  maximum = 0x3ff0;
2701.}
2702.
2703.void CLASS sony_arw_load_raw()
2704.{
2705.  ushort huff[32770];
2706.  static const ushort tab[18] =
2707.  { 0xf11,0xf10,0xe0f,0xd0e,0xc0d,0xb0c,0xa0b,0x90a,0x809,
2708.    0x708,0x607,0x506,0x405,0x304,0x303,0x300,0x202,0x201 };
2709.  int i, c, n, col, row, sum=0;
2710.
2711.  huff[0] = 15;
2712.  for (n=i=0; i < 18; i++)
2713.    FORC(32768 >> (tab[i] >> 8)) huff[++n] = tab[i];
2714.  getbits(-1);
2715.  for (col = raw_width; col--; )
2716.    for (row=0; row < raw_height+1; row+=2) {
2717.      if (row == raw_height) row = 1;
2718.      if ((sum += ljpeg_diff(huff)) >> 12) derror();
2719.      if (row < height) RAW(row,col) = sum;
2720.    }
2721.}
2722.
2723.void CLASS sony_arw2_load_raw()
2724.{
2725.  uchar *data, *dp;
```

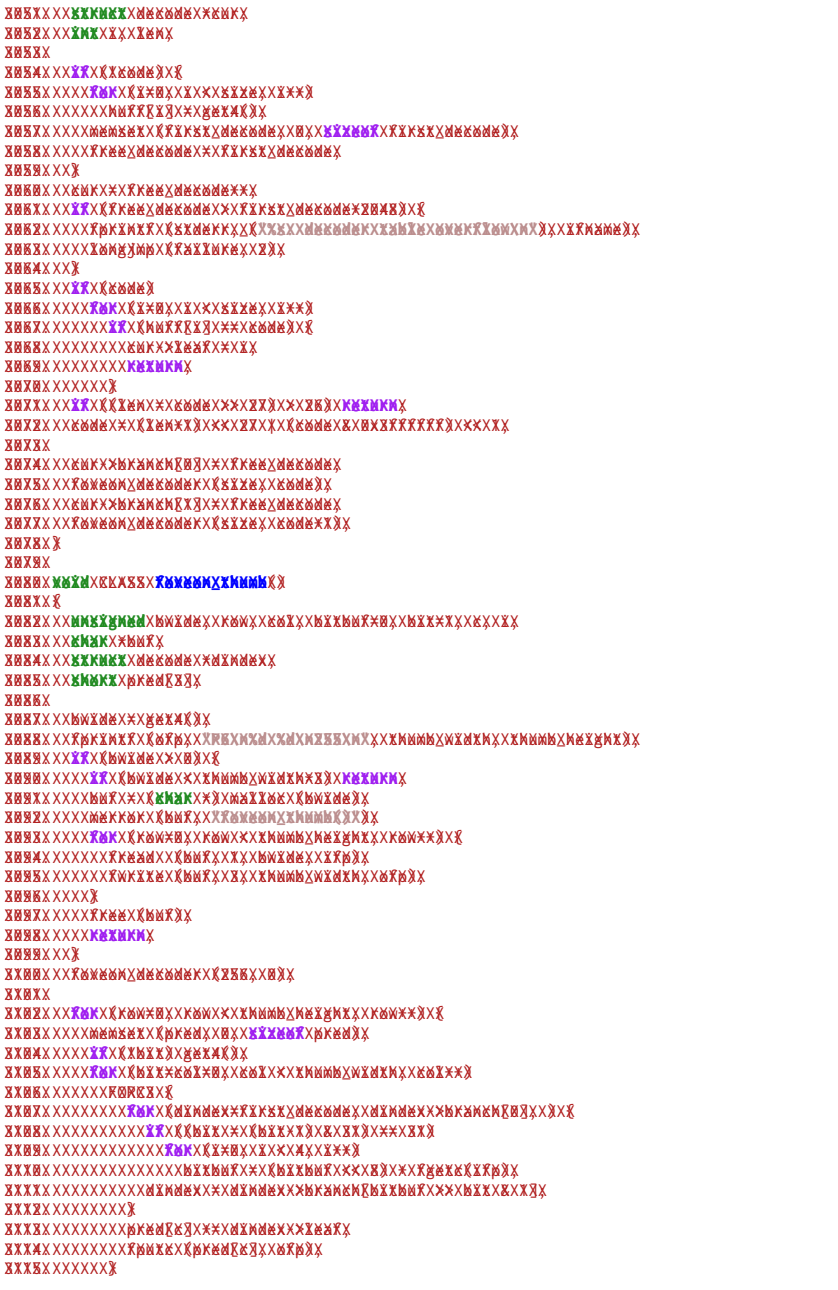
```
2726.  ushort pix[16];
2727.  int row, col, val, max, min, imax, imin, sh, bit, i;
2728.
2729.  data = (uchar *) malloc (raw_width+1);
2730.  merror (data, "sony_arw2_load_raw()");
2731.  for (row=0; row < height; row++) {
2732.    fread (data, 1, raw_width, ifp);
2733.    for (dp=data, col=0; col < raw_width-30; dp+=16) {
2734.      max = 0x7ff & (val = sget4(dp));
2735.      min = 0x7ff & val >> 11;
2736.      imax = 0x0f & val >> 22;
2737.      imin = 0x0f & val >> 26;
2738.      for (sh=0; sh < 4 && 0x80 << sh <= max-min; sh++);
2739.      for (bit=30, i=0; i < 16; i++)
2740.        if      (i == imax) pix[i] = max;
2741.        else if (i == imin) pix[i] = min;
2742.        else {
2743.          pix[i] = ((sget2(dp+(bit >> 3)) >> (bit & 7) & 0x7f) << sh) + min;
2744.          if (pix[i] > 0x7ff) pix[i] = 0x7ff;
2745.          bit += 7;
2746.        }
2747.      for (i=0; i < 16; i++, col+=2)
2748.        RAW(row,col) = curve[pix[i] << 1] >> 2;
2749.      col -= col & 1 ? 1:31;
2750.    }
2751.  }
2752.  free (data);
2753.}
2754.
2755.void CLASS samsung_load_raw()
2756.{
2757.  int row, col, c, i, dir, op[4], len[4];
2758.
2759.  order = 0x4949;
2760.  for (row=0; row < raw_height; row++) {
2761.    fseek (ifp, strip_offset+row*4, SEEK_SET);
2762.    fseek (ifp, data_offset+get4(), SEEK_SET);
2763.    ph1_bits(-1);
2764.    FORC4 len[c] = row < 2 ? 7:4;
2765.    for (col=0; col < raw_width; col+=16) {
2766.      dir = ph1_bits(1);
2767.      FORC4 op[c] = ph1_bits(2);
2768.      FORC4 switch (op[c]) {
2769.        case 3: len[c] = ph1_bits(4);   break;
2770.        case 2: len[c]--;               break;
2771.        case 1: len[c]++;
2772.      }
2773.      for (c=0; c < 16; c+=2) {
2774.        i = len[((c & 1) << 1) | (c >> 3)];
2775.        RAW(row,col+c) = ((signed) ph1_bits(i) << (32-i) >> (32-i)) +
2776.          (dir ? RAW(row+(~c | -2),col+c) : col ? RAW(row,col+(c | -2)) : 128);
2777.        if (c == 14) c = -1;
2778.      }
2779.    }
2780.  }
2781.  for (row=0; row < raw_height-1; row+=2)
2782.    for (col=0; col < raw_width-1; col+=2)
2783.      SWAP (RAW(row,col+1), RAW(row+1,col));
2784.}
2785.
2786.void CLASS samsung2_load_raw()
2787.{
2788.  static const ushort tab[14] =
2789.  { 0x304,0x307,0x206,0x205,0x403,0x600,0x709,
2790.    0x80a,0x90b,0xa0c,0xa0d,0x501,0x408,0x402 };
```

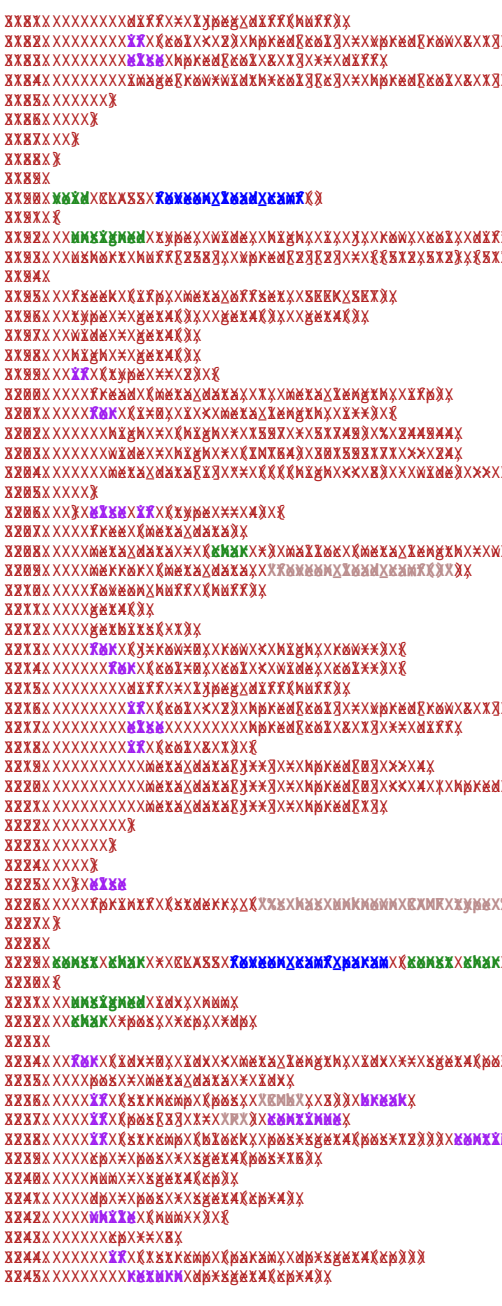
```
  ushort huff[1026], vpred[2][2] = {{0,0},{0,0}}, hpred[2];
  int i, c, n, row, col, diff;

  huff[0] = 10;
  for (n=i=0; i < 14; i++)
    FORC(1024 >> (tab[i] >> 8)) huff[++n] = tab[i];
  getbits(-1);
  for (row=0; row < raw_height; row++)
    for (col=0; col < raw_width; col++) {
      diff = ljpeg_diff (huff);
      if (col < 2) hpred[col] = vpred[row & 1][col] += diff;
      else         hpred[col & 1] += diff;
      RAW(row,col) = hpred[col & 1];
      if (hpred[col & 1] >> tiff_bps) derror();
    }
}

void CLASS samsung3_load_raw()
{
  int opt, init, mag, pmode, row, tab, col, pred, diff, i, c;
  ushort lent[3][2], len[4], *prow[2];

  order = 0x4949;
  fseek (ifp, 9, SEEK_CUR);
  opt = fgetc(ifp);
  init = (get2(),get2());
  for (row=0; row < raw_height; row++) {
    fseek (ifp, (data_offset-ftell(ifp)) & 15, SEEK_CUR);
    ph1_bits(-1);
    mag = 0; pmode = 7;
    FORC(6) ((ushort *)lent)[c] = row < 2 ? 7:4;
    prow[ row & 1] = &RAW(row-1,1-((row & 1) << 1));    // green
    prow[~row & 1] = &RAW(row-2,0);                     // red and blue
    for (tab=0; tab+15 < raw_width; tab+=16) {
      if (~opt & 4 && !(tab & 63)) {
        i = ph1_bits(2);
        mag = i < 3 ? mag-'2'+"204"[i] : ph1_bits(12);
      }
      if (opt & 2)
        pmode = 7 - 4*ph1_bits(1);
      else if (!ph1_bits(1))
        pmode = ph1_bits(3);
      if (opt & 1 || !ph1_bits(1)) {
        FORC4 len[c] = ph1_bits(2);
        FORC4 {
          i = ((row & 1) << 1 | (c & 1)) % 3;
          len[c] = len[c] < 3 ? lent[i][0]-'1'+"120"[len[c]] : ph1_bits(4);
          lent[i][0] = lent[i][1];
          lent[i][1] = len[c];
        }
      }
      FORC(16) {
        col = tab + (((c & 7) << 1)^(c >> 3)^(row & 1));
        pred = (pmode == 7 || row < 2)
             ? (tab ? RAW(row,tab-2+(col & 1)) : init)
             : (prow[col & 1][col-'4'+"0224468"[pmode]] +
                prow[col & 1][col-'4'+"0244668"[pmode]] + 1) >> 1;
        diff = ph1_bits (i = len[c >> 2]);
        if (diff >> (i-1)) diff -= 1 << i;
        diff = diff * (mag*2+1) + mag;
        RAW(row,col) = pred + diff;
      }
    }
  }
}
```

```
2856.
2857.#define HOLE(row) ((holes >> (((row) - raw_height) & 7)) & 1)
2858.
2859./* Kudos to Rich Taylor for figuring out SMaL's compression algorithm. */
2860.void CLASS smal_decode_segment (unsigned seg[2][2], int holes)
2861.{
2862.  uchar hist[3][13] = {
2863.    { 7, 7, 0, 0, 63, 55, 47, 39, 31, 23, 15, 7, 0 },
2864.    { 7, 7, 0, 0, 63, 55, 47, 39, 31, 23, 15, 7, 0 },
2865.    { 3, 3, 0, 0, 63,     47,     31,     15,    0 } };
2866.  int low, high=0xff, carry=0, nbits=8;
2867.  int pix, s, count, bin, next, i, sym[3];
2868.  uchar diff, pred[]={0,0};
2869.  ushort data=0, range=0;
2870.
2871.  fseek (ifp, seg[0][1]+1, SEEK_SET);
2872.  getbits(-1);
2873.  if (seg[1][0] > raw_width*raw_height)
2874.      seg[1][0] = raw_width*raw_height;
2875.  for (pix=seg[0][0]; pix < seg[1][0]; pix++) {
2876.    for (s=0; s < 3; s++) {
2877.      data = data << nbits | getbits(nbits);
2878.      if (carry < 0)
2879.        carry = (nbits += carry+1) < 1 ? nbits-1 : 0;
2880.      while (--nbits >= 0)
2881.        if ((data >> nbits & 0xff) == 0xff) break;
2882.      if (nbits > 0)
2883.          data = ((data & ((1 << (nbits-1)) - 1)) << 1) |
2884.        ((data + (((data & (1 << (nbits-1))) << 1)) & (-1 << nbits));
2885.      if (nbits >= 0) {
2886.        data += getbits(1);
2887.        carry = nbits - 8;
2888.      }
2889.      count = ((((data-range+1) & 0xffff) << 2) - 1) / (high >> 4);
2890.      for (bin=0; hist[s][bin+5] > count; bin++);
2891.                low = hist[s][bin+5] * (high >> 4) >> 2;
2892.      if (bin) high = hist[s][bin+4] * (high >> 4) >> 2;
2893.      high -= low;
2894.      for (nbits=0; high << nbits < 128; nbits++);
2895.      range = (range+low) << nbits;
2896.      high <<= nbits;
2897.      next = hist[s][1];
2898.      if (++hist[s][2] > hist[s][3]) {
2899.        next = (next+1) & hist[s][0];
2900.        hist[s][3] = (hist[s][next+4] - hist[s][next+5]) >> 2;
2901.        hist[s][2] = 1;
2902.      }
2903.      if (hist[s][hist[s][1]+4] - hist[s][hist[s][1]+5] > 1) {
2904.        if (bin < hist[s][1])
2905.          for (i=bin; i < hist[s][1]; i++) hist[s][i+5]--;
2906.        else if (next <= bin)
2907.          for (i=hist[s][1]; i < bin; i++) hist[s][i+5]++;
2908.      }
2909.      hist[s][1] = next;
2910.      sym[s] = bin;
2911.    }
2912.    diff = sym[2] << 5 | sym[1] << 2 | (sym[0] & 3);
2913.    if (sym[0] & 4)
2914.      diff = diff ? -diff : 0x80;
2915.    if (ftell(ifp) + 12 >= seg[1][1])
2916.      diff = 0;
2917.    raw_image[pix] = pred[pix & 1] += diff;
2918.    if (!(pix & 1) && HOLE(pix / raw_width)) pix += 2;
2919.  }
2920.  maximum = 0xff;
```

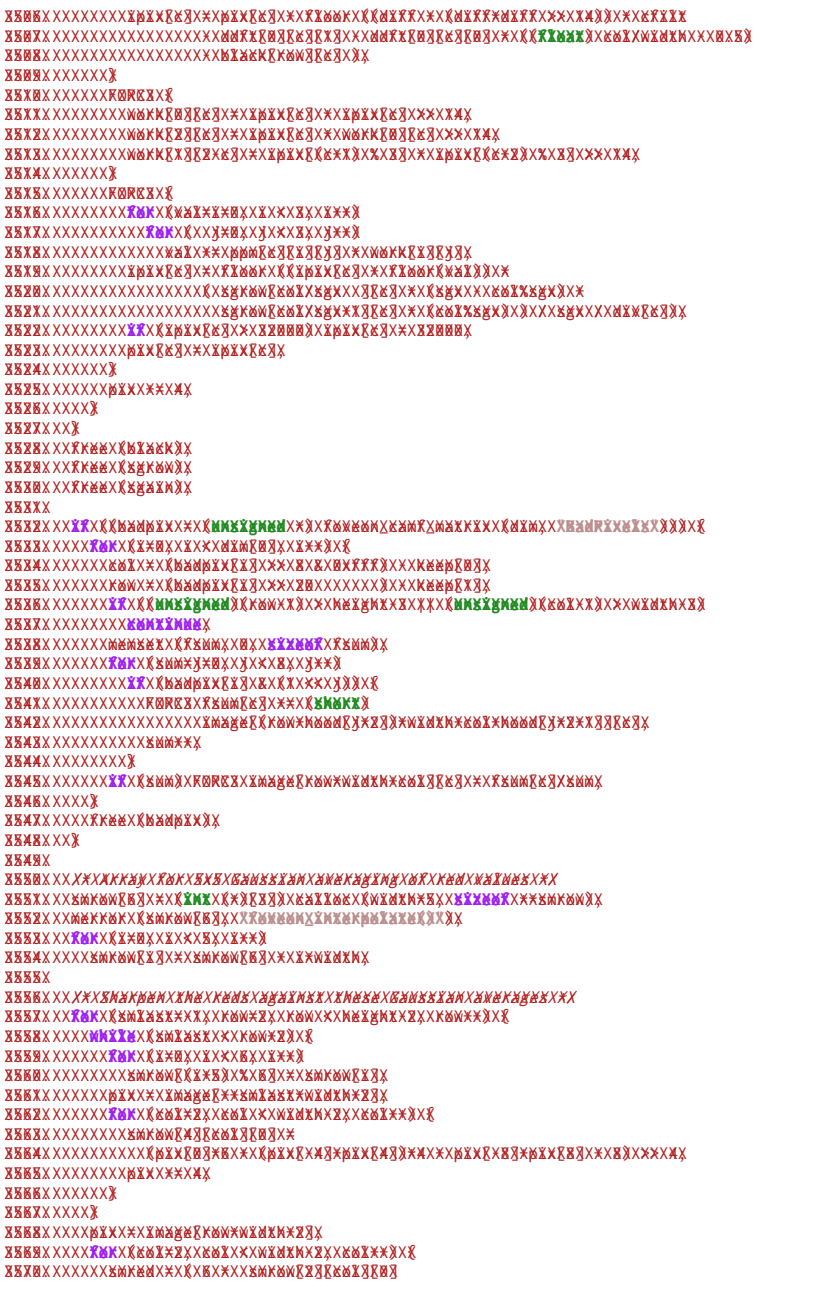
```
2921.}
2922.
2923.void CLASS smal_v6_load_raw()
2924.{
2925.  unsigned seg[2][2];
2926.
2927.  fseek (ifp, 16, SEEK_SET);
2928.  seg[0][0] = 0;
2929.  seg[0][1] = get2();
2930.  seg[1][0] = raw_width * raw_height;
2931.  seg[1][1] = INT_MAX;
2932.  smal_decode_segment (seg, 0);
2933.}
2934.
2935.int CLASS median4 (int *p)
2936.{
2937.  int min, max, sum, i;
2938.
2939.  min = max = sum = p[0];
2940.  for (i=1; i < 4; i++) {
2941.    sum += p[i];
2942.    if (min > p[i]) min = p[i];
2943.    if (max < p[i]) max = p[i];
2944.  }
2945.  return (sum - min - max) >> 1;
2946.}
2947.
2948.void CLASS fill_holes (int holes)
2949.{
2950.  int row, col, val[4];
2951.
2952.  for (row=2; row < height-2; row++) {
2953.    if (!HOLE(row)) continue;
2954.    for (col=1; col < width-1; col+=4) {
2955.      val[0] = RAW(row-1,col-1);
2956.      val[1] = RAW(row-1,col+1);
2957.      val[2] = RAW(row+1,col-1);
2958.      val[3] = RAW(row+1,col+1);
2959.      RAW(row,col) = median4(val);
2960.    }
2961.    for (col=2; col < width-2; col+=4)
2962.      if (HOLE(row-2) || HOLE(row+2))
2963.        RAW(row,col) = (RAW(row,col-2) + RAW(row,col+2)) >> 1;
2964.      else {
2965.        val[0] = RAW(row,col-2);
2966.        val[1] = RAW(row,col+2);
2967.        val[2] = RAW(row-2,col);
2968.        val[3] = RAW(row+2,col);
2969.        RAW(row,col) = median4(val);
2970.      }
2971.  }
2972.}
2973.
2974.void CLASS smal_v9_load_raw()
2975.{
2976.  unsigned seg[256][2], offset, nseg, holes, i;
2977.
2978.  fseek (ifp, 67, SEEK_SET);
2979.  offset = get4();
2980.  nseg = (uchar) fgetc(ifp);
2981.  fseek (ifp, offset, SEEK_SET);
2982.  for (i=0; i < nseg*2; i++)
2983.    ((unsigned *)seg)[i] = get4() + data_offset*(i & 1);
2984.  fseek (ifp, 78, SEEK_SET);
2985.  holes = fgetc(ifp);
```

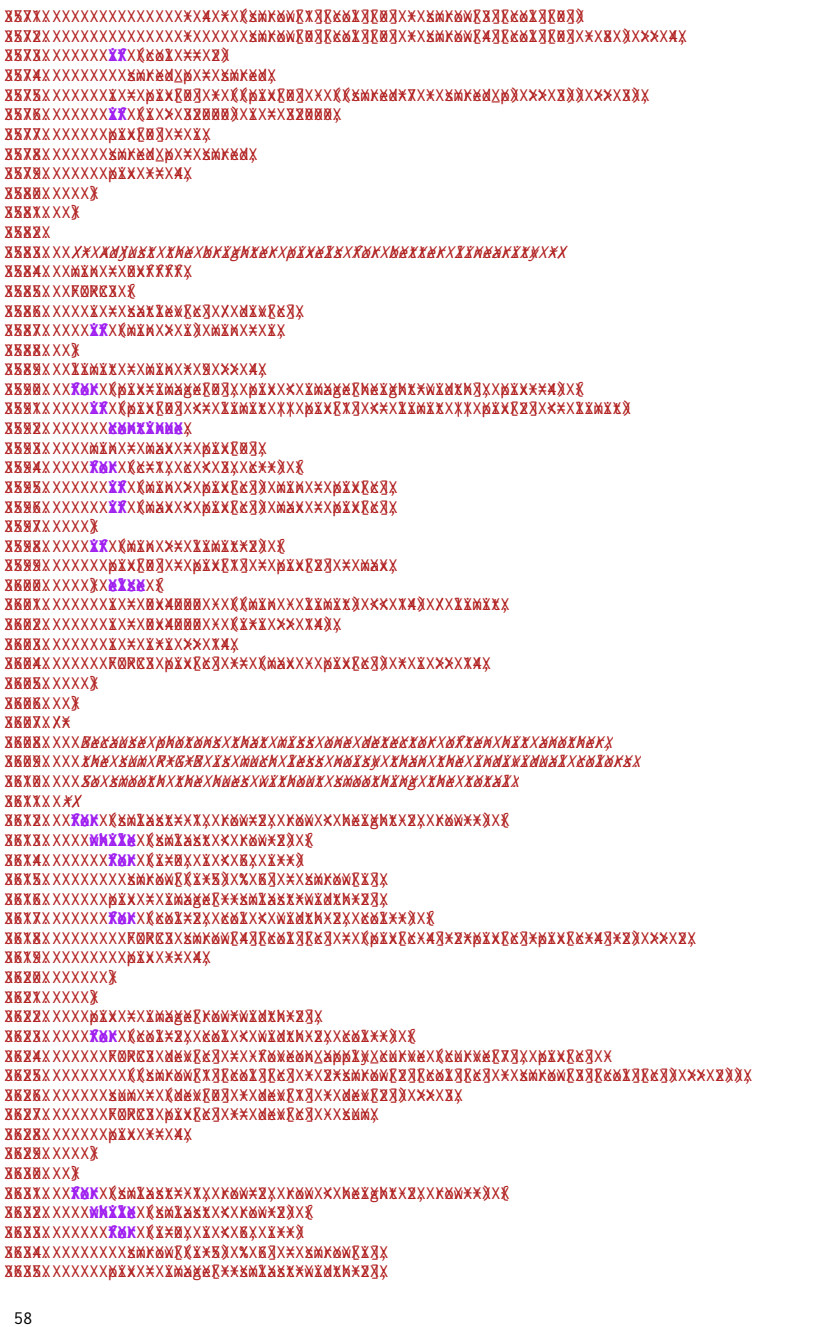
```
  fseek (ifp, 88, SEEK_SET);
  seg[nseg][0] = raw_height * raw_width;
  seg[nseg][1] = get4() + data_offset;
  for (i=0; i < nseg; i++)
    smal_decode_segment (seg+i, holes);
  if (holes) fill_holes (holes);
}

void CLASS redcine_load_raw()
{
#ifndef NO_JASPER
  int c, row, col;
  jas_stream_t *in;
  jas_image_t *jimg;
  jas_matrix_t *jmat;
  jas_seqent_t *data;
  ushort *img, *pix;

  jas_init();
  in = jas_stream_fopen (ifname, "rb");
  jas_stream_seek (in, data_offset+20, SEEK_SET);
  jimg = jas_image_decode (in, -1, 0);
  if (!jimg) longjmp (failure, 3);
  jmat = jas_matrix_create (height/2, width/2);
  merror (jmat, "redcine_load_raw()");
  img = (ushort *) calloc ((height+2), (width+2)*2);
  merror (img, "redcine_load_raw()");
  FORC4 {
    jas_image_readcmpt (jimg, c, 0, 0, width/2, height/2, jmat);
    data = jas_matrix_getref (jmat, 0, 0);
    for (row = c >> 1; row < height; row+=2)
      for (col = c & 1; col < width; col+=2)
        img[(row+1)*(width+2)+col+1] = data[(row/2)*(width/2)+col/2];
  }
  for (col=1; col <= width; col++) {
    img[col] = img[2*(width+2)+col];
    img[(height+1)*(width+2)+col] = img[(height-1)*(width+2)+col];
  }
  for (row=0; row < height+2; row++) {
    img[row*(width+2)] = img[row*(width+2)+2];
    img[(row+1)*(width+2)-1] = img[(row+1)*(width+2)-3];
  }
  for (row=1; row <= height; row++) {
    pix = img + row*(width+2) + (col = 1 + (FC(row,1) & 1));
    for (   ; col <= width; col+=2, pix+=2) {
      c = (((pix[0] - 0x800) << 3) +
        pix[-(width+2)] + pix[width+2] + pix[-1] + pix[1]) >> 2;
      pix[0] = LIM(c,0,4095);
    }
  }
  for (row=0; row < height; row++)
    for (col=0; col < width; col++)
      RAW(row,col) = curve[img[(row+1)*(width+2)+col+1]];
  free (img);
  jas_matrix_destroy (jmat);
  jas_image_destroy (jimg);
  jas_stream_close (in);
#endif
}

/* RESTRICTED code starts here */
```

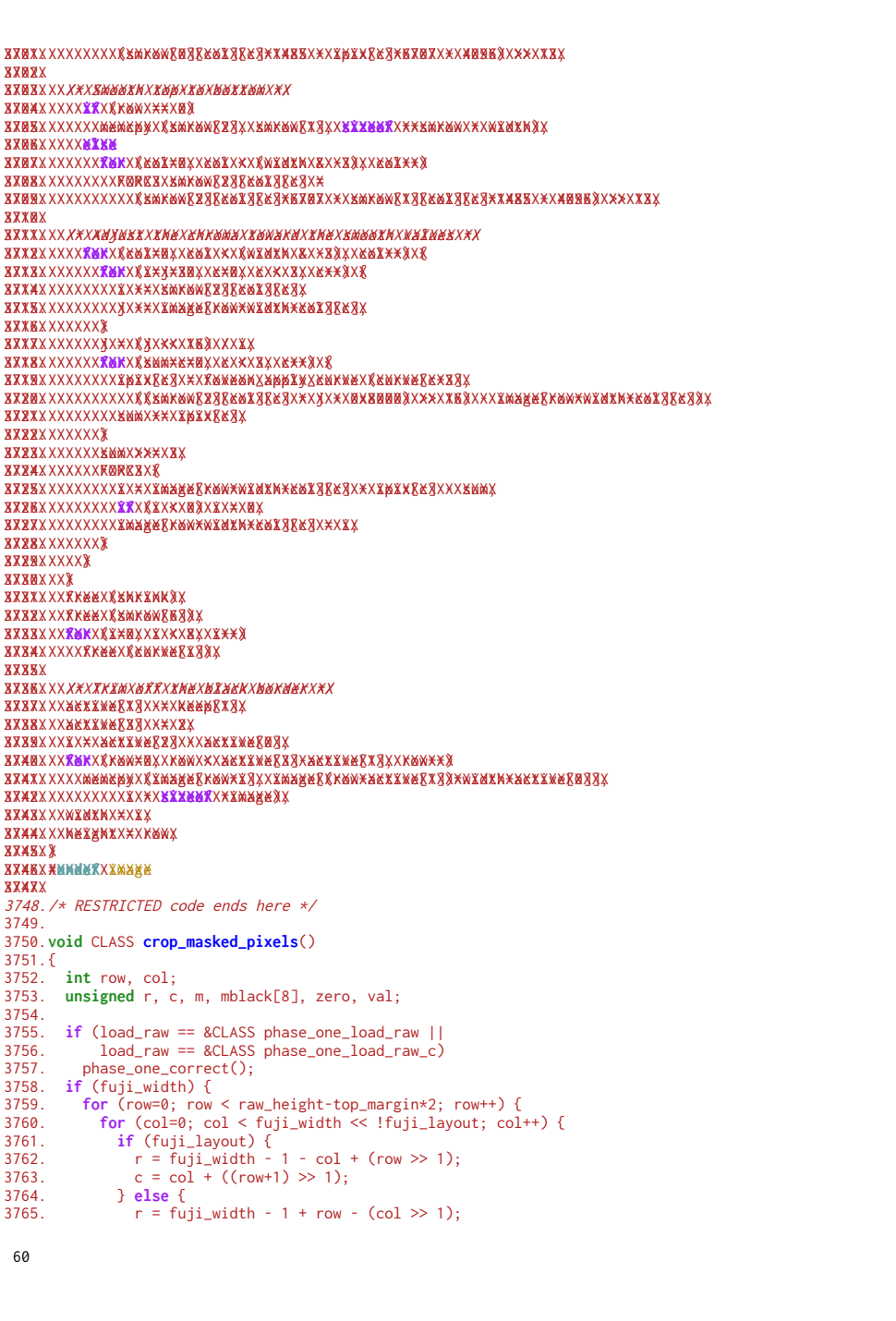




```
xxxxxxxxx
xxxxxxxx
xxxxxxxxxxxxxxx
xxxxxxx
xxxxx
xxxxxxxxxxxxxx
```







```
3748. /* RESTRICTED code ends here */
3749.
3750. void CLASS crop_masked_pixels()
3751. {
3752.   int row, col;
3753.   unsigned r, c, m, mblack[8], zero, val;
3754.
3755.   if (load_raw == &CLASS phase_one_load_raw ||
3756.       load_raw == &CLASS phase_one_load_raw_c)
3757.     phase_one_correct();
3758.   if (fuji_width) {
3759.     for (row=0; row < raw_height-top_margin*2; row++) {
3760.       for (col=0; col < fuji_width << !fuji_layout; col++) {
3761.         if (fuji_layout) {
3762.           r = fuji_width - 1 - col + (row >> 1);
3763.           c = col + ((row+1) >> 1);
3764.         } else {
3765.           r = fuji_width - 1 + row - (col >> 1);
```

```
3766.           c = row + ((col+1) >> 1);
3767.         }
3768.         if (r < height && c < width)
3769.           BAYER(r,c) = RAW(row+top_margin,col+left_margin);
3770.       }
3771.     }
3772.   } else {
3773.     for (row=0; row < height; row++)
3774.       for (col=0; col < width; col++)
3775.         BAYER2(row,col) = RAW(row+top_margin,col+left_margin);
3776.   }
3777.   if (mask[0][3] > 0) goto mask_set;
3778.   if (load_raw == &CLASS canon_load_raw ||
3779.       load_raw == &CLASS lossless_jpeg_load_raw) {
3780.     mask[0][1] = mask[1][1] += 2;
3781.     mask[0][3] -= 2;
3782.     goto sides;
3783.   }
3784.   if (load_raw == &CLASS canon_600_load_raw ||
3785.       load_raw == &CLASS sony_load_raw ||
3786.      (load_raw == &CLASS eight_bit_load_raw && strncmp(model,"DC2",3)) ||
3787.       load_raw == &CLASS kodak_262_load_raw ||
3788.      (load_raw == &CLASS packed_load_raw && (load_flags & 256))) {
3789. sides:
3790.     mask[0][0] = mask[1][0] = top_margin;
3791.     mask[0][2] = mask[1][2] = top_margin+height;
3792.     mask[0][3] += left_margin;
3793.     mask[1][1] += left_margin+width;
3794.     mask[1][3] += raw_width;
3795.   }
3796.   if (load_raw == &CLASS nokia_load_raw) {
3797.     mask[0][2] = top_margin;
3798.     mask[0][3] = width;
3799.   }
3800. mask_set:
3801.   memset (mblack, 0, sizeof mblack);
3802.   for (zero=m=0; m < 8; m++)
3803.     for (row=MAX(mask[m][0],0); row < MIN(mask[m][2],raw_height); row++)
3804.       for (col=MAX(mask[m][1],0); col < MIN(mask[m][3],raw_width); col++) {
3805.         c = FC(row-top_margin,col-left_margin);
3806.         mblack[c] += val = RAW(row,col);
3807.         mblack[4+c]++;
3808.         zero += !val;
3809.       }
3810.   if (load_raw == &CLASS canon_600_load_raw && width < raw_width) {
3811.     black = (mblack[0]+mblack[1]+mblack[2]+mblack[3]) /
3812.             (mblack[4]+mblack[5]+mblack[6]+mblack[7]) - 4;
3813.     canon_600_correct();
3814.   } else if (zero < mblack[4] && mblack[5] && mblack[6] && mblack[7]) {
3815.     FORC4 cblack[c] = mblack[c] / mblack[4+c];
3816.     cblack[4] = cblack[5] = cblack[6] = 0;
3817.   }
3818. }
3819.
3820. void CLASS remove_zeroes()
3821. {
3822.   unsigned row, col, tot, n, r, c;
3823.
3824.   for (row=0; row < height; row++)
3825.     for (col=0; col < width; col++)
3826.       if (BAYER(row,col) == 0) {
3827.         tot = n = 0;
3828.         for (r = row-2; r <= row+2; r++)
3829.           for (c = col-2; c <= col+2; c++)
3830.             if (r < height && c < width &&
```

```
                FC(r,c) == FC(row,col) && BAYER(r,c))
              tot += (n++,BAYER(r,c));
        if (n) BAYER(row,col) = tot/n;
      }
}

/*
   Seach from the current directory up to the root looking for
   a ".badpixels" file, and fix those pixels now.
 */
void CLASS bad_pixels (const char *cfname)
{
  FILE *fp=0;
  char *fname, *cp, line[128];
  int len, time, row, col, r, c, rad, tot, n, fixed=0;

  if (!filters) return;
  if (cfname)
    fp = fopen (cfname, "r");
  else {
    for (len=32 ; ; len *= 2) {
      fname = (char *) malloc (len);
      if (!fname) return;
      if (getcwd (fname, len-16)) break;
      free (fname);
      if (errno != ERANGE) return;
    }
#if defined(WIN32) || defined(DJGPP)
    if (fname[1] == ':')
      memmove (fname, fname+2, len-2);
    for (cp=fname; *cp; cp++)
      if (*cp == '\\') *cp = '/';
#endif
    cp = fname + strlen(fname);
    if (cp[-1] == '/') cp--;
    while (*fname == '/') {
      strcpy (cp, "/.badpixels");
      if ((fp = fopen (fname, "r"))) break;
      if (cp == fname) break;
      while (*--cp != '/');
    }
    free (fname);
  }
  if (!fp) return;
  while (fgets (line, 128, fp)) {
    cp = strchr (line, '#');
    if (cp) *cp = 0;
    if (sscanf (line, "%d %d %d", &col, &row, &time) != 3) continue;
    if ((unsigned) col >= width || (unsigned) row >= height) continue;
    if (time > timestamp) continue;
    for (tot=n=0, rad=1; rad < 3 && n==0; rad++)
      for (r = row-rad; r <= row+rad; r++)
        for (c = col-rad; c <= col+rad; c++)
          if ((unsigned) r < height && (unsigned) c < width &&
                (r != row || c != col) && fcol(r,c) == fcol(row,col)) {
            tot += BAYER2(r,c);
            n++;
          }
    BAYER2(row,col) = tot/n;
    if (verbose) {
      if (!fixed++)
        fprintf (stderr,_("Fixed dead pixels at:"));
      fprintf (stderr, " %d,%d", col, row);
    }
  }
```

```
3896.  if (fixed) fputc ('\n', stderr);
3897.  fclose (fp);
3898.}
3899.
3900.void CLASS subtract (const char *fname)
3901.{
3902.  FILE *fp;
3903.  int dim[3]={0,0,0}, comment=0, number=0, error=0, nd=0, c, row, col;
3904.  ushort *pixel;
3905.
3906.  if (!(fp = fopen (fname, "rb"))) {
3907.    perror (fname);  return;
3908.  }
3909.  if (fgetc(fp) != 'P' || fgetc(fp) != '5') error = 1;
3910.  while (!error && nd < 3 && (c = fgetc(fp)) != EOF) {
3911.    if (c == '#')  comment = 1;
3912.    if (c == '\n') comment = 0;
3913.    if (comment) continue;
3914.    if (isdigit(c)) number = 1;
3915.    if (number) {
3916.      if (isdigit(c)) dim[nd] = dim[nd]*10 + c -'0';
3917.      else if (isspace(c)) {
3918.        number = 0;  nd++;
3919.      } else error = 1;
3920.    }
3921.  }
3922.  if (error || nd < 3) {
3923.    fprintf (stderr,_("%s is not a valid PGM file!\n"), fname);
3924.    fclose (fp);  return;
3925.  } else if (dim[0] != width || dim[1] != height || dim[2] != 65535) {
3926.    fprintf (stderr,_("%s has the wrong dimensions!\n"), fname);
3927.    fclose (fp);  return;
3928.  }
3929.  pixel = (ushort *) calloc (width, sizeof *pixel);
3930.  merror (pixel, "subtract()");
3931.  for (row=0; row < height; row++) {
3932.    fread (pixel, 2, width, fp);
3933.    for (col=0; col < width; col++)
3934.      BAYER(row,col) = MAX (BAYER(row,col) - ntohs(pixel[col]), 0);
3935.  }
3936.  free (pixel);
3937.  fclose (fp);
3938.  memset (cblack, 0, sizeof cblack);
3939.  black = 0;
3940.}
3941.
3942.void CLASS gamma_curve (double pwr, double ts, int mode, int imax)
3943.{
3944.  int i;
3945.  double g[6], bnd[2]={0,0}, r;
3946.
3947.  g[0] = pwr;
3948.  g[1] = ts;
3949.  g[2] = g[3] = g[4] = 0;
3950.  bnd[g[1] >= 1] = 1;
3951.  if (g[1] && (g[1]-1)*(g[0]-1) <= 0) {
3952.    for (i=0; i < 48; i++) {
3953.      g[2] = (bnd[0] + bnd[1])/2;
3954.      if (g[0]) bnd[(pow(g[2]/g[1],-g[0]) - 1)/g[0] - 1/g[2] > -1] = g[2];
3955.      else      bnd[g[2]/exp(1-1/g[2]) < g[1]] = g[2];
3956.    }
3957.    g[3] = g[2] / g[1];
3958.    if (g[0]) g[4] = g[2] * (1/g[0] - 1);
3959.  }
3960.  if (g[0]) g[5] = 1 / (g[1]*SQR(g[3])/2 - g[4]*(1 - g[3]) +
```

```
3961.                (1 - pow(g[3],1+g[0]))*(1 + g[4])/(1 + g[0])) - 1;
3962.  else      g[5] = 1 / (g[1]*SQR(g[3])/2 + 1
3963.                - g[2] - g[3] - g[2]*g[3]*(log(g[3]) - 1)) - 1;
3964.  if (!mode--) {
3965.    memcpy (gamm, g, sizeof gamm);
3966.    return;
3967.  }
3968.  for (i=0; i < 0x10000; i++) {
3969.    curve[i] = 0xffff;
3970.    if ((r = (double) i / imax) < 1)
3971.      curve[i] = 0x10000 * ( mode
3972.        ? (r < g[3] ? r*g[1] : (g[0] ? pow( r,g[0])*(1+g[4])-g[4]    : log(r)*g[2]+1))
3973.        : (r < g[2] ? r/g[1] : (g[0] ? pow((r+g[4])/(1+g[4]),1/g[0]) :
     exp((r-1)/g[2]))));
3974.  }
3975.}
3976.
3977.void CLASS pseudoinverse (double (*in)[3], double (*out)[3], int size)
3978.{
3979.  double work[3][6], num;
3980.  int i, j, k;
3981.
3982.  for (i=0; i < 3; i++) {
3983.    for (j=0; j < 6; j++)
3984.      work[i][j] = j == i+3;
3985.    for (j=0; j < 3; j++)
3986.      for (k=0; k < size; k++)
3987.        work[i][j] += in[k][i] * in[k][j];
3988.  }
3989.  for (i=0; i < 3; i++) {
3990.    num = work[i][i];
3991.    for (j=0; j < 6; j++)
3992.      work[i][j] /= num;
3993.    for (k=0; k < 3; k++) {
3994.      if (k==i) continue;
3995.      num = work[k][i];
3996.      for (j=0; j < 6; j++)
3997.        work[k][j] -= work[i][j] * num;
3998.    }
3999.  }
4000.  for (i=0; i < size; i++)
4001.    for (j=0; j < 3; j++)
4002.      for (out[i][j]=k=0; k < 3; k++)
4003.        out[i][j] += work[j][k+3] * in[i][k];
4004.}
4005.
4006.void CLASS cam_xyz_coeff (float rgb_cam[3][4], double cam_xyz[4][3])
4007.{
4008.  double cam_rgb[4][3], inverse[4][3], num;
4009.  int i, j, k;
4010.
4011.  for (i=0; i < colors; i++)            /* Multiply out XYZ colorspace */
4012.    for (j=0; j < 3; j++)
4013.      for (cam_rgb[i][j] = k=0; k < 3; k++)
4014.        cam_rgb[i][j] += cam_xyz[i][k] * xyz_rgb[k][j];
4015.
4016.  for (i=0; i < colors; i++) {          /* Normalize cam_rgb so that */
4017.    for (num=j=0; j < 3; j++)           /* cam_rgb * (1,1,1) is (1,1,1,1) */
4018.      num += cam_rgb[i][j];
4019.    for (j=0; j < 3; j++)
4020.      cam_rgb[i][j] /= num;
4021.    pre_mul[i] = 1 / num;
4022.  }
4023.  pseudoinverse (cam_rgb, inverse, colors);
4024.  for (i=0; i < 3; i++)
```

```
    for (j=0; j < colors; j++)
      rgb_cam[i][j] = inverse[j][i];
}

#ifdef COLORCHECK
void CLASS colorcheck()
{
#define NSQ 24
// Coordinates of the GretagMacbeth ColorChecker squares
// width, height, 1st_column, 1st_row
  int cut[NSQ][4];                    // you must set these
// ColorChecker Chart under 6500-kelvin illumination
  static const double gmb_xyY[NSQ][3] = {
    { 0.400, 0.350, 10.1 },           // Dark Skin
    { 0.377, 0.345, 35.8 },           // Light Skin
    { 0.247, 0.251, 19.3 },           // Blue Sky
    { 0.337, 0.422, 13.3 },           // Foliage
    { 0.265, 0.240, 24.3 },           // Blue Flower
    { 0.261, 0.343, 43.1 },           // Bluish Green
    { 0.506, 0.407, 30.1 },           // Orange
    { 0.211, 0.175, 12.0 },           // Purplish Blue
    { 0.453, 0.306, 19.8 },           // Moderate Red
    { 0.285, 0.202, 6.6 },            // Purple
    { 0.380, 0.489, 44.3 },           // Yellow Green
    { 0.473, 0.438, 43.1 },           // Orange Yellow
    { 0.187, 0.129, 6.1 },            // Blue
    { 0.305, 0.478, 23.4 },           // Green
    { 0.539, 0.313, 12.0 },           // Red
    { 0.448, 0.470, 59.1 },           // Yellow
    { 0.364, 0.233, 19.8 },           // Magenta
    { 0.196, 0.252, 19.8 },           // Cyan
    { 0.310, 0.316, 90.0 },           // White
    { 0.310, 0.316, 59.1 },           // Neutral 8
    { 0.310, 0.316, 36.2 },           // Neutral 6.5
    { 0.310, 0.316, 19.8 },           // Neutral 5
    { 0.310, 0.316, 9.0 },            // Neutral 3.5
    { 0.310, 0.316, 3.1 } };          // Black
  double gmb_cam[NSQ][4], gmb_xyz[NSQ][3];
  double inverse[NSQ][3], cam_xyz[4][3], balance[4], num;
  int c, i, j, k, sq, row, col, pass, count[4];

  memset (gmb_cam, 0, sizeof gmb_cam);
  for (sq=0; sq < NSQ; sq++) {
    FORCC count[c] = 0;
    for   (row=cut[sq][3]; row < cut[sq][3]+cut[sq][1]; row++)
      for (col=cut[sq][2]; col < cut[sq][2]+cut[sq][0]; col++) {
        c = FC(row,col);
        if (c >= colors) c -= 2;
        gmb_cam[sq][c] += BAYER2(row,col);
        BAYER2(row,col) = black + (BAYER2(row,col)-black)/2;
        count[c]++;
      }
    FORCC gmb_cam[sq][c] = gmb_cam[sq][c]/count[c] - black;
    gmb_xyz[sq][0] = gmb_xyY[sq][2] * gmb_xyY[sq][0] / gmb_xyY[sq][1];
    gmb_xyz[sq][1] = gmb_xyY[sq][2];
    gmb_xyz[sq][2] = gmb_xyY[sq][2] *
                (1 - gmb_xyY[sq][0] - gmb_xyY[sq][1]) / gmb_xyY[sq][1];
  }
  pseudoinverse (gmb_xyz, inverse, NSQ);
  for (pass=0; pass < 2; pass++) {
    for (raw_color = i=0; i < colors; i++)
      for (j=0; j < 3; j++)
        for (cam_xyz[i][j] = k=0; k < NSQ; k++)
          cam_xyz[i][j] += gmb_cam[k][i] * inverse[k][j];
    cam_xyz_coeff (rgb_cam, cam_xyz);
```

```
4090.    FORCC balance[c] = pre_mul[c] * gmb_cam[20][c];
4091.    for (sq=0; sq < NSQ; sq++)
4092.      FORCC gmb_cam[sq][c] *= balance[c];
4093.  }
4094.  if (verbose) {
4095.    printf ("    { \"%s %s\", %d,\n\t{", make, model, black);
4096.    num = 10000 / (cam_xyz[1][0] + cam_xyz[1][1] + cam_xyz[1][2]);
4097.    FORCC for (j=0; j < 3; j++)
4098.      printf ("%c%d", (c | j) ? ',':' ', (int) (cam_xyz[c][j] * num + 0.5));
4099.    puts (" } },");
4100.  }
4101.#undef NSQ
4102.}
4103.#endif
4104.
4105.void CLASS hat_transform (float *temp, float *base, int st, int size, int sc)
4106.{
4107.  int i;
4108.  for (i=0; i < sc; i++)
4109.    temp[i] = 2*base[st*i] + base[st*(sc-i)] + base[st*(i+sc)];
4110.  for (; i+sc < size; i++)
4111.    temp[i] = 2*base[st*i] + base[st*(i-sc)] + base[st*(i+sc)];
4112.  for (; i < size; i++)
4113.    temp[i] = 2*base[st*i] + base[st*(i-sc)] + base[st*(2*size-2-(i+sc))];
4114.}
4115.
4116.void CLASS wavelet_denoise()
4117.{
4118.  float *fimg=0, *temp, thold, mul[2], avg, diff;
4119.  int scale=1, size, lev, hpass, lpass, row, col, nc, c, i, wlast, blk[2];
4120.  ushort *window[4];
4121.  static const float noise[] =
4122.  { 0.8002,0.2735,0.1202,0.0585,0.0291,0.0152,0.0080,0.0044 };
4123.
4124.  if (verbose) fprintf (stderr,_("Wavelet denoising...\n"));
4125.
4126.  while (maximum << scale < 0x10000) scale++;
4127.  maximum <<= --scale;
4128.  black <<= scale;
4129.  FORC4 cblack[c] <<= scale;
4130.  if ((size = iheight*iwidth) < 0x15550000)
4131.    fimg = (float *) malloc ((size*3 + iheight + iwidth) * sizeof *fimg);
4132.  merror (fimg, "wavelet_denoise()");
4133.  temp = fimg + size*3;
4134.  if ((nc = colors) == 3 && filters) nc++;
4135.  FORC(nc) {                    /* denoise R,G1,B,G3 individually */
4136.    for (i=0; i < size; i++)
4137.      fimg[i] = 256 * sqrt(image[i][c] << scale);
4138.    for (hpass=lev=0; lev < 5; lev++) {
4139.      lpass = size*((lev & 1)+1);
4140.      for (row=0; row < iheight; row++) {
4141.        hat_transform (temp, fimg+hpass+row*iwidth, 1, iwidth, 1 << lev);
4142.        for (col=0; col < iwidth; col++)
4143.          fimg[lpass + row*iwidth + col] = temp[col] * 0.25;
4144.      }
4145.      for (col=0; col < iwidth; col++) {
4146.        hat_transform (temp, fimg+lpass+col, iwidth, iheight, 1 << lev);
4147.        for (row=0; row < iheight; row++)
4148.          fimg[lpass + row*iwidth + col] = temp[row] * 0.25;
4149.      }
4150.      thold = threshold * noise[lev];
4151.      for (i=0; i < size; i++) {
4152.        fimg[hpass+i] -= fimg[lpass+i];
4153.        if      (fimg[hpass+i] < -thold) fimg[hpass+i] += thold;
4154.        else if (fimg[hpass+i] >  thold) fimg[hpass+i] -= thold;
```

```
4155.         else     fimg[hpass+i] = 0;
4156.         if (hpass) fimg[i] += fimg[hpass+i];
4157.       }
4158.       hpass = lpass;
4159.     }
4160.     for (i=0; i < size; i++)
4161.       image[i][c] = CLIP(SQR(fimg[i]+fimg[lpass+i])/0x10000);
4162.   }
4163.   if (filters && colors == 3) {  /* pull G1 and G3 closer together */
4164.     for (row=0; row < 2; row++) {
4165.       mul[row] = 0.125 * pre_mul[FC(row+1,0) | 1] / pre_mul[FC(row,0) | 1];
4166.       blk[row] = cblack[FC(row,0) | 1];
4167.     }
4168.     for (i=0; i < 4; i++)
4169.       window[i] = (ushort *) fimg + width*i;
4170.     for (wlast=-1, row=1; row < height-1; row++) {
4171.       while (wlast < row+1) {
4172.         for (wlast++, i=0; i < 4; i++)
4173.           window[(i+3) & 3] = window[i];
4174.         for (col = FC(wlast,1) & 1; col < width; col+=2)
4175.           window[2][col] = BAYER(wlast,col);
4176.       }
4177.       thold = threshold/512;
4178.       for (col = (FC(row,0) & 1)+1; col < width-1; col+=2) {
4179.         avg = ( window[0][col-1] + window[0][col+1] +
4180.                 window[2][col-1] + window[2][col+1] - blk[~row & 1]*4 )
4181.               * mul[row & 1] + (window[1][col] + blk[row & 1]) * 0.5;
4182.         avg = avg < 0 ? 0 : sqrt(avg);
4183.         diff = sqrt(BAYER(row,col)) - avg;
4184.         if      (diff < -thold) diff += thold;
4185.         else if (diff >  thold) diff -= thold;
4186.         else diff = 0;
4187.         BAYER(row,col) = CLIP(SQR(avg+diff) + 0.5);
4188.       }
4189.     }
4190.   }
4191.   free (fimg);
4192. }
4193.
4194. void CLASS scale_colors()
4195. {
4196.   unsigned bottom, right, size, row, col, ur, uc, i, x, y, c, sum[8];
4197.   int val, dark, sat;
4198.   double dsum[8], dmin, dmax;
4199.   float scale_mul[4], fr, fc;
4200.   ushort *img=0, *pix;
4201.
4202.   if (user_mul[0])
4203.     memcpy (pre_mul, user_mul, sizeof pre_mul);
4204.   if (use_auto_wb || (use_camera_wb && cam_mul[0] == -1)) {
4205.     memset (dsum, 0, sizeof dsum);
4206.     bottom = MIN (greybox[1]+greybox[3], height);
4207.     right  = MIN (greybox[0]+greybox[2], width);
4208.     for (row=greybox[1]; row < bottom; row += 8)
4209.       for (col=greybox[0]; col < right; col += 8) {
4210.         memset (sum, 0, sizeof sum);
4211.         for (y=row; y < row+8 && y < bottom; y++)
4212.           for (x=col; x < col+8 && x < right; x++)
4213.             FORC4 {
4214.               if (filters) {
4215.                 c = fcol(y,x);
4216.                 val = BAYER2(y,x);
4217.               } else
4218.                 val = image[y*width+x][c];
4219.               if (val > maximum-25) goto skip_block;
```

```
4220.               if ((val -= cblack[c]) < 0) val = 0;
4221.               sum[c] += val;
4222.               sum[c+4]++;
4223.               if (filters) break;
4224.             }
4225.         FORC(8) dsum[c] += sum[c];
4226. skip_block: ;
4227.       }
4228.     FORC4 if (dsum[c]) pre_mul[c] = dsum[c+4] / dsum[c];
4229.   }
4230.   if (use_camera_wb && cam_mul[0] != -1) {
4231.     memset (sum, 0, sizeof sum);
4232.     for (row=0; row < 8; row++)
4233.       for (col=0; col < 8; col++) {
4234.         c = FC(row,col);
4235.         if ((val = white[row][col] - cblack[c]) > 0)
4236.           sum[c] += val;
4237.         sum[c+4]++;
4238.       }
4239.     if (sum[0] && sum[1] && sum[2] && sum[3])
4240.       FORC4 pre_mul[c] = (float) sum[c+4] / sum[c];
4241.     else if (cam_mul[0] && cam_mul[2])
4242.       memcpy (pre_mul, cam_mul, sizeof pre_mul);
4243.     else
4244.       fprintf (stderr,_("%s: Cannot use camera white balance.\n"), ifname);
4245.   }
4246.   if (pre_mul[1] == 0) pre_mul[1] = 1;
4247.   if (pre_mul[3] == 0) pre_mul[3] = colors < 4 ? pre_mul[1] : 1;
4248.   dark = black;
4249.   sat = maximum;
4250.   if (threshold) wavelet_denoise();
4251.   maximum -= black;
4252.   for (dmin=DBL_MAX, dmax=c=0; c < 4; c++) {
4253.     if (dmin > pre_mul[c])
4254.         dmin = pre_mul[c];
4255.     if (dmax < pre_mul[c])
4256.         dmax = pre_mul[c];
4257.   }
4258.   if (!highlight) dmax = dmin;
4259.   FORC4 scale_mul[c] = (pre_mul[c] /= dmax) * 65535.0 / maximum;
4260.   if (verbose) {
4261.     fprintf (stderr,
4262.       _("Scaling with darkness %d, saturation %d, and\nmultipliers"), dark, sat);
4263.     FORC4 fprintf (stderr, " %f", pre_mul[c]);
4264.     fputc ('\n', stderr);
4265.   }
4266.   if (filters > 1000 && (cblack[4]+1)/2 == 1 && (cblack[5]+1)/2 == 1) {
4267.     FORC4 cblack[FC(c/2,c%2)] +=
4268.         cblack[6 + c/2 % cblack[4] * cblack[5] + c%2 % cblack[5]];
4269.     cblack[4] = cblack[5] = 0;
4270.   }
4271.   size = iheight*iwidth;
4272.   for (i=0; i < size*4; i++) {
4273.     if (!(val = ((ushort *)image)[i])) continue;
4274.     if (cblack[4] && cblack[5])
4275.       val -= cblack[6 + i/4 / iwidth % cblack[4] * cblack[5] +
4276.                        i/4 % iwidth % cblack[5]];
4277.     val -= cblack[i & 3];
4278.     val *= scale_mul[i & 3];
4279.     ((ushort *)image)[i] = CLIP(val);
4280.   }
4281.   if ((aber[0] != 1 || aber[2] != 1) && colors == 3) {
4282.     if (verbose)
4283.       fprintf (stderr,_("Correcting chromatic aberration...\n"));
4284.     for (c=0; c < 4; c+=2) {
```

```
4285.      if (aber[c] == 1) continue;
4286.      img = (ushort *) malloc (size * sizeof *img);
4287.      merror (img, "scale_colors()");
4288.      for (i=0; i < size; i++)
4289.        img[i] = image[i][c];
4290.      for (row=0; row < iheight; row++) {
4291.        ur = fr = (row - iheight*0.5) * aber[c] + iheight*0.5;
4292.        if (ur > iheight-2) continue;
4293.        fr -= ur;
4294.        for (col=0; col < iwidth; col++) {
4295.          uc = fc = (col - iwidth*0.5) * aber[c] + iwidth*0.5;
4296.          if (uc > iwidth-2) continue;
4297.          fc -= uc;
4298.          pix = img + ur*iwidth + uc;
4299.          image[row*iwidth+col][c] =
4300.            (pix[     0]*(1-fc) + pix[       1]*fc) * (1-fr) +
4301.            (pix[iwidth]*(1-fc) + pix[iwidth+1]*fc) * fr;
4302.        }
4303.      }
4304.      free(img);
4305.    }
4306.  }
4307.}
4308.
4309.void CLASS pre_interpolate()
4310.{
4311.  ushort (*img)[4];
4312.  int row, col, c;
4313.
4314.  if (shrink) {
4315.    if (half_size) {
4316.      height = iheight;
4317.      width  = iwidth;
4318.      if (filters == 9) {
4319.        for (row=0; row < 3; row++)
4320.          for (col=1; col < 4; col++)
4321.            if (!(image[row*width+col][0] | image[row*width+col][2]))
4322.              goto break2;  break2:
4323.        for ( ; row < height; row+=3)
4324.          for (col=(col-1)%3+1; col < width-1; col+=3) {
4325.            img = image + row*width+col;
4326.            for (c=0; c < 3; c+=2)
4327.              img[0][c] = (img[-1][c] + img[1][c]) >> 1;
4328.          }
4329.      }
4330.    } else {
4331.      img = (ushort (*)[4]) calloc (height, width*sizeof *img);
4332.      merror (img, "pre_interpolate()");
4333.      for (row=0; row < height; row++)
4334.        for (col=0; col < width; col++) {
4335.          c = fcol(row,col);
4336.          img[row*width+col][c] = image[(row >> 1)*iwidth+(col >> 1)][c];
4337.        }
4338.      free (image);
4339.      image = img;
4340.      shrink = 0;
4341.    }
4342.  }
4343.  if (filters > 1000 && colors == 3) {
4344.    mix_green = four_color_rgb ^ half_size;
4345.    if (four_color_rgb | half_size) colors++;
4346.    else {
4347.      for (row = FC(1,0) >> 1; row < height; row+=2)
4348.        for (col = FC(row,1) & 1; col < width; col+=2)
4349.          image[row*width+col][1] = image[row*width+col][3];
```

```
4350.      filters &= ~((filters & 0x55555555) << 1);
4351.    }
4352.  }
4353.  if (half_size) filters = 0;
4354.}
4355.
4356.void CLASS border_interpolate (int border)
4357.{
4358.  unsigned row, col, y, x, f, c, sum[8];
4359.
4360.  for (row=0; row < height; row++)
4361.    for (col=0; col < width; col++) {
4362.      if (col==border && row >= border && row < height-border)
4363.        col = width-border;
4364.      memset (sum, 0, sizeof sum);
4365.      for (y=row-1; y != row+2; y++)
4366.        for (x=col-1; x != col+2; x++)
4367.          if (y < height && x < width) {
4368.            f = fcol(y,x);
4369.            sum[f] += image[y*width+x][f];
4370.            sum[f+4]++;
4371.          }
4372.      f = fcol(row,col);
4373.      FORCC if (c != f && sum[c+4])
4374.        image[row*width+col][c] = sum[c] / sum[c+4];
4375.    }
4376.}
4377.
4378.void CLASS lin_interpolate()
4379.{
4380.  int code[16][16][32], size=16, *ip, sum[4];
4381.  int f, c, i, x, y, row, col, shift, color;
4382.  ushort *pix;
4383.
4384.  if (verbose) fprintf (stderr,_("Bilinear interpolation...\n"));
4385.  if (filters == 9) size = 6;
4386.  border_interpolate(1);
4387.  for (row=0; row < size; row++)
4388.    for (col=0; col < size; col++) {
4389.      ip = code[row][col]+1;
4390.      f = fcol(row,col);
4391.      memset (sum, 0, sizeof sum);
4392.      for (y=-1; y <= 1; y++)
4393.        for (x=-1; x <= 1; x++) {
4394.          shift = (y==0) + (x==0);
4395.          color = fcol(row+y,col+x);
4396.          if (color == f) continue;
4397.          *ip++ = (width*y + x)*4 + color;
4398.          *ip++ = shift;
4399.          *ip++ = color;
4400.          sum[color] += 1 << shift;
4401.        }
4402.      code[row][col][0] = (ip - code[row][col]) / 3;
4403.      FORCC
4404.        if (c != f) {
4405.          *ip++ = c;
4406.          *ip++ = 256 / sum[c];
4407.        }
4408.    }
4409.  for (row=1; row < height-1; row++)
4410.    for (col=1; col < width-1; col++) {
4411.      pix = image[row*width+col];
4412.      ip = code[row % size][col % size];
4413.      memset (sum, 0, sizeof sum);
4414.      for (i=*ip++; i--; ip+=3)
```

```
4415.         sum[ip[2]] += pix[ip[0]] << ip[1];
4416.       for (i=colors; --i; ip+=2)
4417.         pix[ip[0]] = sum[ip[0]] * ip[1] >> 8;
4418.     }
4419. }
4420.
4421. /*
4422.    This algorithm is officially called:
4423.
4424.    "Interpolation using a Threshold-based variable number of gradients"
4425.
4426.    described in http://scien.stanford.edu/pages/labsite/1999/psych221/projects/99/
      tingchen/algodep/vargra.html
4427.
4428.    I've extended the basic idea to work with non-Bayer filter arrays.
4429.    Gradients are numbered clockwise from NW=0 to W=7.
4430.  */
4431. void CLASS vng_interpolate()
4432. {
4433.   static const signed char *cp, terms[] = {
4434.     -2,-2,+0,-1,0,0x01, -2,-2,+0,+0,1,0x01, -2,-1,-1,+0,0,0x01,
4435.     -2,-1,+0,-1,0,0x02, -2,-1,+0,+0,0,0x03, -2,-1,+0,+1,1,0x01,
4436.     -2,+0,+0,-1,0,0x06, -2,+0,+0,+0,1,0x02, -2,+0,+0,+1,0,0x03,
4437.     -2,+1,-1,+0,0,0x04, -2,+1,+0,-1,1,0x04, -2,+1,+0,+0,0,0x06,
4438.     -2,+1,+0,+1,0,0x02, -2,+2,+0,+0,1,0x04, -2,+2,+0,+1,0,0x04,
4439.     -1,-2,-1,+0,0,0x80, -1,-2,+0,-1,0,0x01, -1,-2,+1,-1,0,0x01,
4440.     -1,-2,+1,+0,1,0x01, -1,-1,-1,+1,0,0x88, -1,-1,+1,-2,0,0x40,
4441.     -1,-1,+1,-1,0,0x22, -1,-1,+1,+0,0,0x33, -1,-1,+1,+1,1,0x11,
4442.     -1,+0,-1,+2,0,0x08, -1,+0,+0,-1,0,0x44, -1,+0,+0,+1,0,0x11,
4443.     -1,+0,+1,-2,1,0x40, -1,+0,+1,-1,0,0x66, -1,+0,+1,+0,1,0x22,
4444.     -1,+0,+1,+1,0,0x33, -1,+0,+1,+2,1,0x10, -1,+1,+1,-1,1,0x44,
4445.     -1,+1,+1,+0,0,0x66, -1,+1,+1,+1,0,0x22, -1,+1,+1,+2,0,0x10,
4446.     -1,+2,+0,+1,0,0x04, -1,+2,+1,+0,1,0x04, -1,+2,+1,+1,0,0x04,
4447.     +0,-2,+0,+0,1,0x80, +0,-1,+0,+1,1,0x88, +0,-1,+1,-2,0,0x40,
4448.     +0,-1,+1,+0,0,0x11, +0,-1,+2,-2,0,0x40, +0,-1,+2,-1,0,0x20,
4449.     +0,-1,+2,+0,0,0x30, +0,-1,+2,+1,1,0x10, +0,+0,+0,+2,1,0x08,
4450.     +0,+0,+2,-2,1,0x40, +0,+0,+2,-1,0,0x60, +0,+0,+2,+0,1,0x20,
4451.     +0,+0,+2,+1,0,0x30, +0,+0,+2,+2,1,0x10, +0,+1,+1,+0,0,0x44,
4452.     +0,+1,+1,+2,0,0x10, +0,+1,+2,-1,1,0x40, +0,+1,+2,+0,0,0x60,
4453.     +0,+1,+2,+1,0,0x20, +0,+1,+2,+2,0,0x10, +1,-2,+1,+0,0,0x80,
4454.     +1,-1,+1,+1,0,0x88, +1,+0,+1,+2,0,0x08, +1,+0,+2,-1,0,0x40,
4455.     +1,+0,+2,+1,0,0x10
4456.   }, chood[] = { -1,-1, -1,0, -1,+1, 0,+1, +1,+1, +1,0, +1,-1, 0,-1 };
4457.   ushort (*brow[5])[4], *pix;
4458.   int prow=8, pcol=2, *ip, *code[16][16], gval[8], gmin, gmax, sum[4];
4459.   int row, col, x, y, x1, x2, y1, y2, t, weight, grads, color, diag;
4460.   int g, diff, thold, num, c;
4461.
4462.   lin_interpolate();
4463.   if (verbose) fprintf (stderr,_("VNG interpolation...\n"));
4464.
4465.   if (filters == 1) prow = pcol = 16;
4466.   if (filters == 9) prow = pcol =  6;
4467.   ip = (int *) calloc (prow*pcol, 1280);
4468.   merror (ip, "vng_interpolate()");
4469.   for (row=0; row < prow; row++)                /* Precalculate for VNG */
4470.     for (col=0; col < pcol; col++) {
4471.       code[row][col] = ip;
4472.       for (cp=terms, t=0; t < 64; t++) {
4473.         y1 = *cp++;  x1 = *cp++;
4474.         y2 = *cp++;  x2 = *cp++;
4475.         weight = *cp++;
4476.         grads = *cp++;
4477.         color = fcol(row+y1,col+x1);
4478.         if (fcol(row+y2,col+x2) != color) continue;
```

```
4479.         diag = (fcol(row,col+1) == color && fcol(row+1,col) == color) ? 2:1;
4480.         if (abs(y1-y2) == diag && abs(x1-x2) == diag) continue;
4481.         *ip++ = (y1*width + x1)*4 + color;
4482.         *ip++ = (y2*width + x2)*4 + color;
4483.         *ip++ = weight;
4484.         for (g=0; g < 8; g++)
4485.           if (grads & 1<<g) *ip++ = g;
4486.         *ip++ = -1;
4487.       }
4488.       *ip++ = INT_MAX;
4489.       for (cp=chood, g=0; g < 8; g++) {
4490.         y = *cp++;  x = *cp++;
4491.         *ip++ = (y*width + x) * 4;
4492.         color = fcol(row,col);
4493.         if (fcol(row+y,col+x) != color && fcol(row+y*2,col+x*2) == color)
4494.           *ip++ = (y*width + x) * 8 + color;
4495.         else
4496.           *ip++ = 0;
4497.       }
4498.     }
4499.   brow[4] = (ushort (*)[4]) calloc (width*3, sizeof **brow);
4500.   merror (brow[4], "vng_interpolate()");
4501.   for (row=0; row < 3; row++)
4502.     brow[row] = brow[4] + row*width;
4503.   for (row=2; row < height-2; row++) {          /* Do VNG interpolation */
4504.     for (col=2; col < width-2; col++) {
4505.       pix = image[row*width+col];
4506.       ip = code[row % prow][col % pcol];
4507.       memset (gval, 0, sizeof gval);
4508.       while ((g = ip[0]) != INT_MAX) {          /* Calculate gradients */
4509.         diff = ABS(pix[g] - pix[ip[1]]) << ip[2];
4510.         gval[ip[3]] += diff;
4511.         ip += 5;
4512.         if ((g = ip[-1]) == -1) continue;
4513.         gval[g] += diff;
4514.         while ((g = *ip++) != -1)
4515.           gval[g] += diff;
4516.       }
4517.       ip++;
4518.       gmin = gmax = gval[0];                    /* Choose a threshold */
4519.       for (g=1; g < 8; g++) {
4520.         if (gmin > gval[g]) gmin = gval[g];
4521.         if (gmax < gval[g]) gmax = gval[g];
4522.       }
4523.       if (gmax == 0) {
4524.         memcpy (brow[2][col], pix, sizeof *image);
4525.         continue;
4526.       }
4527.       thold = gmin + (gmax >> 1);
4528.       memset (sum, 0, sizeof sum);
4529.       color = fcol(row,col);
4530.       for (num=g=0; g < 8; g++,ip+=2) {         /* Average the neighbors */
4531.         if (gval[g] <= thold) {
4532.           FORCC
4533.             if (c == color && ip[1])
4534.               sum[c] += (pix[c] + pix[ip[1]]) >> 1;
4535.             else
4536.               sum[c] += pix[ip[0] + c];
4537.           num++;
4538.         }
4539.       }
4540.       FORCC {                                   /* Save to buffer */
4541.         t = pix[color];
4542.         if (c != color)
4543.           t += (sum[c] - sum[color]) / num;
```

```
4544.         brow[2][col][c] = CLIP(t);
4545.       }
4546.     }
4547.     if (row > 3)                                /* Write buffer to image */
4548.       memcpy (image[(row-2)*width+2], brow[0]+2, (width-4)*sizeof *image);
4549.     for (g=0; g < 4; g++)
4550.       brow[(g-1) & 3] = brow[g];
4551.   }
4552.   memcpy (image[(row-2)*width+2], brow[0]+2, (width-4)*sizeof *image);
4553.   memcpy (image[(row-1)*width+2], brow[1]+2, (width-4)*sizeof *image);
4554.   free (brow[4]);
4555.   free (code[0][0]);
4556. }
4557.
4558. /*
4559.    Patterned Pixel Grouping Interpolation by Alain Desbiolles
4560. */
4561. void CLASS ppg_interpolate()
4562. {
4563.   int dir[5] = { 1, width, -1, -width, 1 };
4564.   int row, col, diff[2], guess[2], c, d, i;
4565.   ushort (*pix)[4];
4566.
4567.   border_interpolate(3);
4568.   if (verbose) fprintf (stderr,_("PPG interpolation...\n"));
4569.
4570. /*  Fill in the green layer with gradients and pattern recognition: */
4571.   for (row=3; row < height-3; row++)
4572.     for (col=3+(FC(row,3) & 1), c=FC(row,col); col < width-3; col+=2) {
4573.       pix = image + row*width+col;
4574.       for (i=0; (d=dir[i]) > 0; i++) {
4575.         guess[i] = (pix[-d][1] + pix[0][c] + pix[d][1]) * 2
4576.                    - pix[-2*d][c] - pix[2*d][c];
4577.         diff[i] = ( ABS(pix[-2*d][c] - pix[ 0][c]) +
4578.                     ABS(pix[ 2*d][c] - pix[ 0][c]) +
4579.                     ABS(pix[  -d][1] - pix[ d][1]) ) * 3 +
4580.                   ( ABS(pix[ 3*d][1] - pix[ d][1]) +
4581.                     ABS(pix[-3*d][1] - pix[-d][1]) ) * 2;
4582.       }
4583.       d = dir[i = diff[0] > diff[1]];
4584.       pix[0][1] = ULIM(guess[i] >> 2, pix[d][1], pix[-d][1]);
4585.     }
4586. /*  Calculate red and blue for each green pixel:                */
4587.   for (row=1; row < height-1; row++)
4588.     for (col=1+(FC(row,2) & 1), c=FC(row,col+1); col < width-1; col+=2) {
4589.       pix = image + row*width+col;
4590.       for (i=0; (d=dir[i]) > 0; c=2-c, i++)
4591.         pix[0][c] = CLIP((pix[-d][c] + pix[d][c] + 2*pix[0][1]
4592.                           - pix[-d][1] - pix[d][1]) >> 1);
4593.     }
4594. /*  Calculate blue for red pixels and vice versa:               */
4595.   for (row=1; row < height-1; row++)
4596.     for (col=1+(FC(row,1) & 1), c=2-FC(row,col); col < width-1; col+=2) {
4597.       pix = image + row*width+col;
4598.       for (i=0; (d=dir[i]+dir[i+1]) > 0; i++) {
4599.         diff[i] = ABS(pix[-d][c] - pix[d][c]) +
4600.                   ABS(pix[-d][1] - pix[0][1]) +
4601.                   ABS(pix[ d][1] - pix[0][1]);
4602.         guess[i] = pix[-d][c] + pix[d][c] + 2*pix[0][1]
4603.                  - pix[-d][1] - pix[d][1];
4604.       }
4605.       if (diff[0] != diff[1])
4606.         pix[0][c] = CLIP(guess[diff[0] > diff[1]] >> 1);
4607.       else
4608.         pix[0][c] = CLIP((guess[0]+guess[1]) >> 2);
```

```
4609.    }
4610.}
4611.
4612.void CLASS cielab (ushort rgb[3], short lab[3])
4613.{
4614.  int c, i, j, k;
4615.  float r, xyz[3];
4616.  static float cbrt[0x10000], xyz_cam[3][4];
4617.
4618.  if (!rgb) {
4619.    for (i=0; i < 0x10000; i++) {
4620.      r = i / 65535.0;
4621.      cbrt[i] = r > 0.008856 ? pow(r,1/3.0) : 7.787*r + 16/116.0;
4622.    }
4623.    for (i=0; i < 3; i++)
4624.      for (j=0; j < colors; j++)
4625.        for (xyz_cam[i][j] = k=0; k < 3; k++)
4626.          xyz_cam[i][j] += xyz_rgb[i][k] * rgb_cam[k][j] / d65_white[i];
4627.    return;
4628.  }
4629.  xyz[0] = xyz[1] = xyz[2] = 0.5;
4630.  FORCC {
4631.    xyz[0] += xyz_cam[0][c] * rgb[c];
4632.    xyz[1] += xyz_cam[1][c] * rgb[c];
4633.    xyz[2] += xyz_cam[2][c] * rgb[c];
4634.  }
4635.  xyz[0] = cbrt[CLIP((int) xyz[0])];
4636.  xyz[1] = cbrt[CLIP((int) xyz[1])];
4637.  xyz[2] = cbrt[CLIP((int) xyz[2])];
4638.  lab[0] = 64 * (116 * xyz[1] - 16);
4639.  lab[1] = 64 * 500 * (xyz[0] - xyz[1]);
4640.  lab[2] = 64 * 200 * (xyz[1] - xyz[2]);
4641.}
4642.
4643.#define TS 512          /* Tile Size */
4644.#define fcol(row,col) xtrans[(row+6) % 6][(col+6) % 6]
4645.
4646./*
4647.   Frank Markesteijn's algorithm for Fuji X-Trans sensors
4648. */
4649.void CLASS xtrans_interpolate (int passes)
4650.{
4651.  int c, d, f, g, h, i, v, ng, row, col, top, left, mrow, mcol;
4652.  int val, ndir, pass, hm[8], avg[4], color[3][8];
4653.  static const short orth[12] = { 1,0,0,1,-1,0,0,-1,1,0,0,1 },
4654.        patt[2][16] = { { 0,1,0,-1,2,0,-1,0,1,1,1,-1,0,0,0,0 },
4655.                        { 0,1,0,-2,1,0,-2,0,1,1,-2,-2,1,-1,-1,1 } },
4656.        dir[4] = { 1,TS,TS+1,TS-1 };
4657.  short allhex[3][3][2][8], *hex;
4658.  ushort min, max, sgrow, sgcol;
4659.  ushort (*rgb)[TS][TS][3], (*rix)[3], (*pix)[4];
4660.   short (*lab)    [TS][3], (*lix)[3];
4661.   float (*drv)[TS][TS], diff[6], tr;
4662.   char (*homo)[TS][TS], *buffer;
4663.
4664.  if (verbose)
4665.    fprintf (stderr,_("%d-pass X-Trans interpolation...\n"), passes);
4666.
4667.  cielab (0,0);
4668.  ndir = 4 << (passes > 1);
4669.  buffer = (char *) malloc (TS*TS*(ndir*11+6));
4670.  merror (buffer, "xtrans_interpolate()");
4671.  rgb  = (ushort(*)[TS][TS][3]) buffer;
4672.  lab  = (short (*)    [TS][3])(buffer + TS*TS*(ndir*6));
4673.  drv  = (float (*)[TS][TS])   (buffer + TS*TS*(ndir*6+6));
```

```
4674.  homo = (char  (*)[TS][TS])   (buffer + TS*TS*(ndir*10+6));
4675.
4676./* Map a green hexagon around each non-green pixel and vice versa:      */
4677.  for (row=0; row < 3; row++)
4678.    for (col=0; col < 3; col++)
4679.      for (ng=d=0; d < 10; d+=2) {
4680.        g = fcol(row,col) == 1;
4681.        if (fcol(row+orth[d],col+orth[d+2]) == 1) ng=0; else ng++;
4682.        if (ng == 4) { sgrow = row; sgcol = col; }
4683.        if (ng == g+1) FORC(8) {
4684.          v = orth[d  ]*patt[g][c*2] + orth[d+1]*patt[g][c*2+1];
4685.          h = orth[d+2]*patt[g][c*2] + orth[d+3]*patt[g][c*2+1];
4686.          allhex[row][col][0][c^(g*2 & d)] = h + v*width;
4687.          allhex[row][col][1][c^(g*2 & d)] = h + v*TS;
4688.        }
4689.      }
4690.
4691./* Set green1 and green3 to the minimum and maximum allowed values:     */
4692.  for (row=2; row < height-2; row++)
4693.    for (min=~(max=0), col=2; col < width-2; col++) {
4694.      if (fcol(row,col) == 1 && (min=~(max=0))) continue;
4695.      pix = image + row*width + col;
4696.      hex = allhex[row % 3][col % 3][0];
4697.      if (!max) FORC(6) {
4698.        val = pix[hex[c]][1];
4699.        if (min > val) min = val;
4700.        if (max < val) max = val;
4701.      }
4702.      pix[0][1] = min;
4703.      pix[0][3] = max;
4704.      switch ((row-sgrow) % 3) {
4705.        case 1: if (row < height-3) { row++; col--; } break;
4706.        case 2: if ((min=~(max=0)) && (col+=2) < width-3 && row > 2) row--;
4707.      }
4708.    }
4709.
4710.  for (top=3; top < height-19; top += TS-16)
4711.    for (left=3; left < width-19; left += TS-16) {
4712.      mrow = MIN (top+TS, height-3);
4713.      mcol = MIN (left+TS, width-3);
4714.      for (row=top; row < mrow; row++)
4715.        for (col=left; col < mcol; col++)
4716.          memcpy (rgb[0][row-top][col-left], image[row*width+col], 6);
4717.      FORC3 memcpy (rgb[c+1], rgb[0], sizeof *rgb);
4718.
4719./* Interpolate green horizontally, vertically, and along both diagonals: */
4720.      for (row=top; row < mrow; row++)
4721.        for (col=left; col < mcol; col++) {
4722.          if ((f = fcol(row,col)) == 1) continue;
4723.          pix = image + row*width + col;
4724.          hex = allhex[row % 3][col % 3][0];
4725.          color[1][0] = 174 * (pix[  hex[1]][1] + pix[  hex[0]][1]) -
4726.                         46 * (pix[2*hex[1]][1] + pix[2*hex[0]][1]);
4727.          color[1][1] = 223 *  pix[  hex[3]][1] + pix[  hex[2]][1] * 33 +
4728.                         92 * (pix[      0 ][f] - pix[ -hex[2]][f]);
4729.          FORC(2) color[1][2+c] =
4730.                164 * pix[hex[4+c]][1] + 92 * pix[-2*hex[4+c]][1] + 33 *
4731.                (2*pix[0][f] - pix[3*hex[4+c]][f] - pix[-3*hex[4+c]][f]);
4732.          FORC4 rgb[c^!((row-sgrow) % 3)][row-top][col-left][1] =
4733.                LIM(color[1][c] >> 8,pix[0][1],pix[0][3]);
4734.        }
4735.
4736.      for (pass=0; pass < passes; pass++) {
4737.        if (pass == 1)
4738.          memcpy (rgb+=4, buffer, 4*sizeof *rgb);
```

```
4739.
4740./* Recalculate green from interpolated values of closer pixels: */
4741.        if (pass) {
4742.          for (row=top+2; row < mrow-2; row++)
4743.            for (col=left+2; col < mcol-2; col++) {
4744.              if ((f = fcol(row,col)) == 1) continue;
4745.              pix = image + row*width + col;
4746.              hex = allhex[row % 3][col % 3][1];
4747.              for (d=3; d < 6; d++) {
4748.                rix = &rgb[(d-2)^!((row-sgrow) % 3)][row-top][col-left];
4749.                val = rix[-2*hex[d]][1] + 2*rix[hex[d]][1]
4750.                    - rix[-2*hex[d]][f] - 2*rix[hex[d]][f] + 3*rix[0][f];
4751.                rix[0][1] = LIM(val/3,pix[0][1],pix[0][3]);
4752.              }
4753.            }
4754.        }
4755.
4756./* Interpolate red and blue values for solitary green pixels:   */
4757.        for (row=(top-sgrow+4)/3*3+sgrow; row < mrow-2; row+=3)
4758.          for (col=(left-sgcol+4)/3*3+sgcol; col < mcol-2; col+=3) {
4759.            rix = &rgb[0][row-top][col-left];
4760.            h = fcol(row,col+1);
4761.            memset (diff, 0, sizeof diff);
4762.            for (i=1, d=0; d < 6; d++, i^=TS^1, h^=2) {
4763.              for (c=0; c < 2; c++, h^=2) {
4764.                g = 2*rix[0][1] - rix[i<<c][1] - rix[-i<<c][1];
4765.                color[h][d] = g + rix[i<<c][h] + rix[-i<<c][h];
4766.                if (d > 1)
4767.                  diff[d] += SQR (rix[i<<c][1] - rix[-i<<c][1]
4768.                                - rix[i<<c][h] + rix[-i<<c][h]) + SQR(g);
4769.              }
4770.              if (d > 1 && (d & 1))
4771.                if (diff[d-1] < diff[d])
4772.                  FORC(2) color[c*2][d] = color[c*2][d-1];
4773.              if (d < 2 || (d & 1)) {
4774.                FORC(2) rix[0][c*2] = CLIP(color[c*2][d]/2);
4775.                rix += TS*TS;
4776.              }
4777.            }
4778.          }
4779.
4780./* Interpolate red for blue pixels and vice versa:              */
4781.        for (row=top+3; row < mrow-3; row++)
4782.          for (col=left+3; col < mcol-3; col++) {
4783.            if ((f = 2-fcol(row,col)) == 1) continue;
4784.            rix = &rgb[0][row-top][col-left];
4785.            c = (row-sgrow) % 3 ? TS:1;
4786.            h = 3 * (c ^ TS ^ 1);
4787.            for (d=0; d < 4; d++, rix += TS*TS) {
4788.              i = d > 1 || ((d ^ c) & 1) ||
4789.                 ((ABS(rix[0][1]-rix[c][1])+ABS(rix[0][1]-rix[-c][1])) <
4790.                2*(ABS(rix[0][1]-rix[h][1])+ABS(rix[0][1]-rix[-h][1]))) ? c:h;
4791.              rix[0][f] = CLIP((rix[i][f] + rix[-i][f] +
4792.                  2*rix[0][1] - rix[i][1] - rix[-i][1])/2);
4793.            }
4794.          }
4795.
4796./* Fill in red and blue for 2x2 blocks of green:               */
4797.        for (row=top+2; row < mrow-2; row++) if ((row-sgrow) % 3)
4798.          for (col=left+2; col < mcol-2; col++) if ((col-sgcol) % 3) {
4799.            rix = &rgb[0][row-top][col-left];
4800.            hex = allhex[row % 3][col % 3][1];
4801.            for (d=0; d < ndir; d+=2, rix += TS*TS)
4802.              if (hex[d] + hex[d+1]) {
4803.                g = 3*rix[0][1] - 2*rix[hex[d]][1] - rix[hex[d+1]][1];
```

```
4804.                 for (c=0; c < 4; c+=2) rix[0][c] =
4805.                         CLIP((g + 2*rix[hex[d]][c] + rix[hex[d+1]][c])/3);
4806.               } else {
4807.                 g = 2*rix[0][1] - rix[hex[d]][1] - rix[hex[d+1]][1];
4808.                 for (c=0; c < 4; c+=2) rix[0][c] =
4809.                         CLIP((g + rix[hex[d]][c] + rix[hex[d+1]][c])/2);
4810.               }
4811.           }
4812.       }
4813.       rgb = (ushort(*)[TS][TS][3]) buffer;
4814.       mrow -= top;
4815.       mcol -= left;
4816.
4817./* Convert to CIELab and differentiate in all directions:       */
4818.       for (d=0; d < ndir; d++) {
4819.         for (row=2; row < mrow-2; row++)
4820.           for (col=2; col < mcol-2; col++)
4821.             cielab (rgb[d][row][col], lab[row][col]);
4822.         for (f=dir[d & 3],row=3; row < mrow-3; row++)
4823.           for (col=3; col < mcol-3; col++) {
4824.             lix = &lab[row][col];
4825.             g = 2*lix[0][0] - lix[f][0] - lix[-f][0];
4826.             drv[d][row][col] = SQR(g)
4827.               + SQR((2*lix[0][1] - lix[f][1] - lix[-f][1] + g*500/232))
4828.               + SQR((2*lix[0][2] - lix[f][2] - lix[-f][2] - g*500/580));
4829.           }
4830.       }
4831.
4832./* Build homogeneity maps from the derivatives:                */
4833.       memset(homo, 0, ndir*TS*TS);
4834.       for (row=4; row < mrow-4; row++)
4835.         for (col=4; col < mcol-4; col++) {
4836.           for (tr=FLT_MAX, d=0; d < ndir; d++)
4837.             if (tr > drv[d][row][col])
4838.                 tr = drv[d][row][col];
4839.           tr *= 8;
4840.           for (d=0; d < ndir; d++)
4841.             for (v=-1; v <= 1; v++)
4842.               for (h=-1; h <= 1; h++)
4843.                 if (drv[d][row+v][col+h] <= tr)
4844.                   homo[d][row][col]++;
4845.         }
4846.
4847./* Average the most homogenous pixels for the final result:     */
4848.       if (height-top < TS+4) mrow = height-top+2;
4849.       if (width-left < TS+4) mcol = width-left+2;
4850.       for (row = MIN(top,8); row < mrow-8; row++)
4851.         for (col = MIN(left,8); col < mcol-8; col++) {
4852.           for (d=0; d < ndir; d++)
4853.             for (hm[d]=0, v=-2; v <= 2; v++)
4854.               for (h=-2; h <= 2; h++)
4855.                 hm[d] += homo[d][row+v][col+h];
4856.           for (d=0; d < ndir-4; d++)
4857.             if (hm[d] < hm[d+4]) hm[d  ] = 0; else
4858.             if (hm[d] > hm[d+4]) hm[d+4] = 0;
4859.           for (max=hm[0],d=1; d < ndir; d++)
4860.             if (max < hm[d]) max = hm[d];
4861.           max -= max >> 3;
4862.           memset (avg, 0, sizeof avg);
4863.           for (d=0; d < ndir; d++)
4864.             if (hm[d] >= max) {
4865.               FORC3 avg[c] += rgb[d][row][col][c];
4866.               avg[3]++;
4867.             }
4868.           FORC3 image[(row+top)*width+col+left][c] = avg[c]/avg[3];
```

```
4869.        }
4870.    }
4871.  free(buffer);
4872.  border_interpolate(8);
4873.}
4874.#undef fcol
4875.
4876./*
4877.   Adaptive Homogeneity-Directed interpolation is based on
4878.   the work of Keigo Hirakawa, Thomas Parks, and Paul Lee.
4879. */
4880.void CLASS ahd_interpolate()
4881.{
4882.  int i, j, top, left, row, col, tr, tc, c, d, val, hm[2];
4883.  static const int dir[4] = { -1, 1, -TS, TS };
4884.  unsigned ldiff[2][4], abdiff[2][4], leps, abeps;
4885.  ushort (*rgb)[TS][TS][3], (*rix)[3], (*pix)[4];
4886.   short (*lab)[TS][TS][3], (*lix)[3];
4887.   char (*homo)[TS][TS], *buffer;
4888.
4889.  if (verbose) fprintf (stderr,_("AHD interpolation...\n"));
4890.
4891.  cielab (0,0);
4892.  border_interpolate(5);
4893.  buffer = (char *) malloc (26*TS*TS);
4894.  merror (buffer, "ahd_interpolate()");
4895.  rgb  = (ushort(*)[TS][TS][3]) buffer;
4896.  lab  = (short (*)[TS][TS][3])(buffer + 12*TS*TS);
4897.  homo = (char  (*)[TS][TS])   (buffer + 24*TS*TS);
4898.
4899.  for (top=2; top < height-5; top += TS-6)
4900.    for (left=2; left < width-5; left += TS-6) {
4901.
4902./*  Interpolate green horizontally and vertically:              */
4903.      for (row=top; row < top+TS && row < height-2; row++) {
4904.        col = left + (FC(row,left) & 1);
4905.        for (c = FC(row,col); col < left+TS && col < width-2; col+=2) {
4906.          pix = image + row*width+col;
4907.          val = ((pix[-1][1] + pix[0][c] + pix[1][1]) * 2
4908.                - pix[-2][c] - pix[2][c]) >> 2;
4909.          rgb[0][row-top][col-left][1] = ULIM(val,pix[-1][1],pix[1][1]);
4910.          val = ((pix[-width][1] + pix[0][c] + pix[width][1]) * 2
4911.                - pix[-2*width][c] - pix[2*width][c]) >> 2;
4912.          rgb[1][row-top][col-left][1] = ULIM(val,pix[-width][1],pix[width][1]);
4913.        }
4914.      }
4915./*  Interpolate red and blue, and convert to CIELab:            */
4916.      for (d=0; d < 2; d++)
4917.        for (row=top+1; row < top+TS-1 && row < height-3; row++)
4918.          for (col=left+1; col < left+TS-1 && col < width-3; col++) {
4919.            pix = image + row*width+col;
4920.            rix = &rgb[d][row-top][col-left];
4921.            lix = &lab[d][row-top][col-left];
4922.            if ((c = 2 - FC(row,col)) == 1) {
4923.              c = FC(row+1,col);
4924.              val = pix[0][1] + (( pix[-1][2-c] + pix[1][2-c]
4925.                                 - rix[-1][1] - rix[1][1] ) >> 1);
4926.              rix[0][2-c] = CLIP(val);
4927.              val = pix[0][1] + (( pix[-width][c] + pix[width][c]
4928.                                 - rix[-TS][1] - rix[TS][1] ) >> 1);
4929.            } else
4930.              val = rix[0][1] + (( pix[-width-1][c] + pix[-width+1][c]
4931.                                 + pix[+width-1][c] + pix[+width+1][c]
4932.                                 - rix[-TS-1][1] - rix[-TS+1][1]
4933.                                 - rix[+TS-1][1] - rix[+TS+1][1] + 1) >> 2);
```

```
4934.             rix[0][c] = CLIP(val);
4935.             c = FC(row,col);
4936.             rix[0][c] = pix[0][c];
4937.             cielab (rix[0],lix[0]);
4938.           }
4939./*  Build homogeneity maps from the CIELab images:               */
4940.       memset (homo, 0, 2*TS*TS);
4941.       for (row=top+2; row < top+TS-2 && row < height-4; row++) {
4942.         tr = row-top;
4943.         for (col=left+2; col < left+TS-2 && col < width-4; col++) {
4944.           tc = col-left;
4945.           for (d=0; d < 2; d++) {
4946.             lix = &lab[d][tr][tc];
4947.             for (i=0; i < 4; i++) {
4948.                ldiff[d][i] = ABS(lix[0][0]-lix[dir[i]][0]);
4949.               abdiff[d][i] = SQR(lix[0][1]-lix[dir[i]][1])
4950.                            + SQR(lix[0][2]-lix[dir[i]][2]);
4951.             }
4952.           }
4953.           leps = MIN(MAX(ldiff[0][0],ldiff[0][1]),
4954.                      MAX(ldiff[1][2],ldiff[1][3]));
4955.           abeps = MIN(MAX(abdiff[0][0],abdiff[0][1]),
4956.                       MAX(abdiff[1][2],abdiff[1][3]));
4957.           for (d=0; d < 2; d++)
4958.             for (i=0; i < 4; i++)
4959.               if (ldiff[d][i] <= leps && abdiff[d][i] <= abeps)
4960.                 homo[d][tr][tc]++;
4961.         }
4962.       }
4963./*  Combine the most homogenous pixels for the final result:    */
4964.       for (row=top+3; row < top+TS-3 && row < height-5; row++) {
4965.         tr = row-top;
4966.         for (col=left+3; col < left+TS-3 && col < width-5; col++) {
4967.           tc = col-left;
4968.           for (d=0; d < 2; d++)
4969.             for (hm[d]=0, i=tr-1; i <= tr+1; i++)
4970.               for (j=tc-1; j <= tc+1; j++)
4971.                 hm[d] += homo[d][i][j];
4972.           if (hm[0] != hm[1])
4973.             FORC3 image[row*width+col][c] = rgb[hm[1] > hm[0]][tr][tc][c];
4974.           else
4975.             FORC3 image[row*width+col][c] =
4976.                 (rgb[0][tr][tc][c] + rgb[1][tr][tc][c]) >> 1;
4977.         }
4978.       }
4979.     }
4980.   free (buffer);
4981. }
4982. #undef TS
4983.
4984. void CLASS median_filter()
4985. {
4986.   ushort (*pix)[4];
4987.   int pass, c, i, j, k, med[9];
4988.   static const uchar opt[] =    /* Optimal 9-element median search */
4989.   { 1,2, 4,5, 7,8, 0,1, 3,4, 6,7, 1,2, 4,5, 7,8,
4990.     0,3, 5,8, 4,7, 3,6, 1,4, 2,5, 4,7, 4,2, 6,4, 4,2 };
4991.
4992.   for (pass=1; pass <= med_passes; pass++) {
4993.     if (verbose)
4994.       fprintf (stderr,_("Median filter pass %d...\n"), pass);
4995.     for (c=0; c < 3; c+=2) {
4996.       for (pix = image; pix < image+width*height; pix++)
4997.         pix[0][3] = pix[0][c];
4998.       for (pix = image+width; pix < image+width*(height-1); pix++) {
```

```
4999.         if ((pix-image+1) % width < 2) continue;
5000.         for (k=0, i = -width; i <= width; i += width)
5001.           for (j = i-1; j <= i+1; j++)
5002.             med[k++] = pix[j][3] - pix[j][1];
5003.         for (i=0; i < sizeof opt; i+=2)
5004.           if     (med[opt[i]] > med[opt[i+1]])
5005.             SWAP (med[opt[i]] , med[opt[i+1]]);
5006.         pix[0][c] = CLIP(med[4] + pix[0][1]);
5007.       }
5008.     }
5009.   }
5010.}
5011.
5012.void CLASS blend_highlights()
5013.{
5014.  int clip=INT_MAX, row, col, c, i, j;
5015.  static const float trans[2][4][4] =
5016.  { { { 1,1,1 }, { 1.7320508,-1.7320508,0 }, { -1,-1,2 } },
5017.    { { 1,1,1,1 }, { 1,-1,1,-1 }, { 1,1,-1,-1 }, { 1,-1,-1,1 } } };
5018.  static const float itrans[2][4][4] =
5019.  { { { 1,0.8660254,-0.5 }, { 1,-0.8660254,-0.5 }, { 1,0,1 } },
5020.    { { 1,1,1,1 }, { 1,-1,1,-1 }, { 1,1,-1,-1 }, { 1,-1,-1,1 } } };
5021.  float cam[2][4], lab[2][4], sum[2], chratio;
5022.
5023.  if ((unsigned) (colors-3) > 1) return;
5024.  if (verbose) fprintf (stderr,_("Blending highlights...\n"));
5025.  FORCC if (clip > (i = 65535*pre_mul[c])) clip = i;
5026.  for (row=0; row < height; row++)
5027.    for (col=0; col < width; col++) {
5028.      FORCC if (image[row*width+col][c] > clip) break;
5029.      if (c == colors) continue;
5030.      FORCC {
5031.        cam[0][c] = image[row*width+col][c];
5032.        cam[1][c] = MIN(cam[0][c],clip);
5033.      }
5034.      for (i=0; i < 2; i++) {
5035.        FORCC for (lab[i][c]=j=0; j < colors; j++)
5036.          lab[i][c] += trans[colors-3][c][j] * cam[i][j];
5037.        for (sum[i]=0,c=1; c < colors; c++)
5038.          sum[i] += SQR(lab[i][c]);
5039.      }
5040.      chratio = sqrt(sum[1]/sum[0]);
5041.      for (c=1; c < colors; c++)
5042.        lab[0][c] *= chratio;
5043.      FORCC for (cam[0][c]=j=0; j < colors; j++)
5044.        cam[0][c] += itrans[colors-3][c][j] * lab[0][j];
5045.      FORCC image[row*width+col][c] = cam[0][c] / colors;
5046.    }
5047.}
5048.
5049.#define SCALE (4 >> shrink)
5050.void CLASS recover_highlights()
5051.{
5052.  float *map, sum, wgt, grow;
5053.  int hsat[4], count, spread, change, val, i;
5054.  unsigned high, wide, mrow, mcol, row, col, kc, c, d, y, x;
5055.  ushort *pixel;
5056.  static const signed char dir[8][2] =
5057.    { {-1,-1}, {-1,0}, {-1,1}, {0,1}, {1,1}, {1,0}, {1,-1}, {0,-1} };
5058.
5059.  if (verbose) fprintf (stderr,_("Rebuilding highlights...\n"));
5060.
5061.  grow = pow (2, 4-highlight);
5062.  FORCC hsat[c] = 32000 * pre_mul[c];
5063.  for (kc=0, c=1; c < colors; c++)
```

```
5064.    if (pre_mul[kc] < pre_mul[c]) kc = c;
5065.  high = height / SCALE;
5066.  wide =  width / SCALE;
5067.  map = (float *) calloc (high, wide*sizeof *map);
5068.  merror (map, "recover_highlights()");
5069.  FORCC if (c != kc) {
5070.    memset (map, 0, high*wide*sizeof *map);
5071.    for (mrow=0; mrow < high; mrow++)
5072.      for (mcol=0; mcol < wide; mcol++) {
5073.        sum = wgt = count = 0;
5074.        for (row = mrow*SCALE; row < (mrow+1)*SCALE; row++)
5075.          for (col = mcol*SCALE; col < (mcol+1)*SCALE; col++) {
5076.            pixel = image[row*width+col];
5077.            if (pixel[c] / hsat[c] == 1 && pixel[kc] > 24000) {
5078.              sum += pixel[c];
5079.              wgt += pixel[kc];
5080.              count++;
5081.            }
5082.          }
5083.        if (count == SCALE*SCALE)
5084.          map[mrow*wide+mcol] = sum / wgt;
5085.      }
5086.    for (spread = 32/grow; spread--; ) {
5087.      for (mrow=0; mrow < high; mrow++)
5088.        for (mcol=0; mcol < wide; mcol++) {
5089.          if (map[mrow*wide+mcol]) continue;
5090.          sum = count = 0;
5091.          for (d=0; d < 8; d++) {
5092.            y = mrow + dir[d][0];
5093.            x = mcol + dir[d][1];
5094.            if (y < high && x < wide && map[y*wide+x] > 0) {
5095.              sum  += (1 + (d & 1)) * map[y*wide+x];
5096.              count += 1 + (d & 1);
5097.            }
5098.          }
5099.          if (count > 3)
5100.            map[mrow*wide+mcol] = - (sum+grow) / (count+grow);
5101.        }
5102.      for (change=i=0; i < high*wide; i++)
5103.        if (map[i] < 0) {
5104.          map[i] = -map[i];
5105.          change = 1;
5106.        }
5107.      if (!change) break;
5108.    }
5109.    for (i=0; i < high*wide; i++)
5110.      if (map[i] == 0) map[i] = 1;
5111.    for (mrow=0; mrow < high; mrow++)
5112.      for (mcol=0; mcol < wide; mcol++) {
5113.        for (row = mrow*SCALE; row < (mrow+1)*SCALE; row++)
5114.          for (col = mcol*SCALE; col < (mcol+1)*SCALE; col++) {
5115.            pixel = image[row*width+col];
5116.            if (pixel[c] / hsat[c] > 1) {
5117.              val = pixel[kc] * map[mrow*wide+mcol];
5118.              if (pixel[c] < val) pixel[c] = CLIP(val);
5119.            }
5120.          }
5121.      }
5122.  }
5123.  free (map);
5124.}
5125.#undef SCALE
5126.
5127.void CLASS tiff_get (unsigned base,
5128.        unsigned *tag, unsigned *type, unsigned *len, unsigned *save)
```

```
5129.{
5130.  *tag  = get2();
5131.  *type = get2();
5132.  *len  = get4();
5133.  *save = ftell(ifp) + 4;
5134.  if (*len > ("11124811248484"[*type < 14 ? *type:0]-'0') > 4)
5135.    fseek (ifp, get4()+base, SEEK_SET);
5136.}
5137.
5138.void CLASS parse_thumb_note (int base, unsigned toff, unsigned tlen)
5139.{
5140.  unsigned entries, tag, type, len, save;
5141.
5142.  entries = get2();
5143.  while (entries--) {
5144.    tiff_get (base, &tag, &type, &len, &save);
5145.    if (tag == toff) thumb_offset = get4()+base;
5146.    if (tag == tlen) thumb_length = get4();
5147.    fseek (ifp, save, SEEK_SET);
5148.  }
5149.}
5150.
5151.int CLASS parse_tiff_ifd (int base);
5152.
5153.void CLASS parse_makernote (int base, int uptag)
5154.{
5155.  static const uchar xlat[2][256] = {
5156.  { 0xc1,0xbf,0x6d,0x0d,0x59,0xc5,0x13,0x9d,0x83,0x61,0x6b,0x4f,0xc7,0x7f,0x3d,0x3d,
5157.    0x53,0x59,0xe3,0xc7,0xe9,0x2f,0x95,0xa7,0x95,0x1f,0xdf,0x7f,0x2b,0x29,0xc7,0x0d,
5158.    0xdf,0x07,0xef,0x71,0x89,0x3d,0x13,0x3d,0x3b,0x13,0xfb,0x0d,0x89,0xc1,0x65,0x1f,
5159.    0xb3,0x0d,0x6b,0x29,0xe3,0xfb,0xef,0xa3,0x6b,0x47,0x7f,0x95,0x35,0xa7,0x47,0x4f,
5160.    0xc7,0xf1,0x59,0x95,0x35,0x11,0x29,0x61,0xf1,0x3d,0xb3,0x2b,0x0d,0x43,0x89,0xc1,
5161.    0x9d,0x9d,0x89,0x65,0xf1,0xe9,0xdf,0xbf,0x3d,0x7f,0x53,0x97,0xe5,0xe9,0x95,0x17,
5162.    0x1d,0x3d,0x8b,0xfb,0xc7,0xe3,0x67,0xa7,0x07,0xf1,0x71,0xa7,0x53,0xb5,0x29,0x89,
5163.    0xe5,0x2b,0xa7,0x17,0x29,0xe9,0x4f,0xc5,0x65,0x6d,0x6b,0xef,0x0d,0x89,0x49,0x2f,
5164.    0xb3,0x43,0x53,0x65,0x1d,0x49,0xa3,0x13,0x89,0x59,0xef,0x6b,0xef,0x65,0x1d,0x0b,
5165.    0x59,0x13,0xe3,0x4f,0x9d,0xb3,0x29,0x43,0x2b,0x07,0x1d,0x95,0x59,0x59,0x47,0xfb,
5166.    0xe5,0xe9,0x61,0x47,0x2f,0x35,0x7f,0x17,0x7f,0xef,0x7f,0x95,0x95,0x71,0xd3,0xa3,
5167.    0x0b,0x71,0xa3,0xad,0x0b,0x3b,0xb5,0xfb,0xa3,0xbf,0x4f,0x83,0x1d,0xad,0xe9,0x2f,
5168.    0x71,0x65,0xa3,0xe5,0x07,0x35,0x3d,0x0d,0xb5,0xe9,0xe5,0x47,0x3b,0x9d,0xef,0x35,
5169.    0xa3,0xbf,0xb3,0xdf,0x53,0xd3,0x97,0x53,0x49,0x71,0x07,0x35,0x61,0x71,0x2f,0x43,
5170.    0x2f,0x11,0xdf,0x17,0x97,0xfb,0x95,0x3b,0x7f,0x6b,0xd3,0x25,0xbf,0xad,0xc7,0xc5,
5171.    0xc5,0xb5,0x8b,0xef,0x2f,0xd3,0x07,0x6b,0x25,0x49,0x95,0x25,0x49,0x6d,0x71,0xc7 },
5172.  { 0xa7,0xbc,0xc9,0xad,0x91,0xdf,0x85,0xe5,0xd4,0x78,0xd5,0x17,0x46,0x7c,0x29,0x4c,
5173.    0x4d,0x03,0xe9,0x25,0x68,0x11,0x86,0xb3,0xbd,0xf7,0x6f,0x61,0x22,0xa2,0x26,0x34,
5174.    0x2a,0xbe,0x1e,0x46,0x14,0x68,0x9d,0x44,0x18,0xc2,0x40,0xf4,0x7e,0x5f,0x1b,0xad,
5175.    0x0b,0x94,0xb6,0x67,0xb4,0x0b,0xe1,0xea,0x95,0x9c,0x66,0xdc,0xe7,0x5d,0x6c,0x05,
5176.    0xda,0xd5,0xdf,0x7a,0xef,0xf6,0xdb,0x1f,0x82,0x4c,0xc0,0x68,0x47,0xa1,0xbd,0xee,
5177.    0x39,0x50,0x56,0x4a,0xdd,0xdf,0xa5,0xf8,0xc6,0xda,0xca,0x90,0xca,0x01,0x42,0x9d,
5178.    0x8b,0x0c,0x73,0x43,0x75,0x05,0x94,0xde,0x24,0xb3,0x80,0x34,0xe5,0x2c,0xdc,0x9b,
5179.    0x3f,0xca,0x33,0x45,0xd0,0xdb,0x5f,0xf5,0x52,0xc3,0x21,0xda,0xe2,0x22,0x72,0x6b,
5180.    0x3e,0xd0,0x5b,0xa8,0x87,0x8c,0x06,0x5d,0x0f,0xdd,0x09,0x19,0x93,0xd0,0xb9,0xfc,
5181.    0x8b,0x0f,0x84,0x60,0x33,0x1c,0x9b,0x45,0xf1,0xf0,0xa3,0x94,0x3a,0x12,0x77,0x33,
5182.    0x4d,0x44,0x78,0x28,0x3c,0x9e,0xfd,0x65,0x57,0x16,0x94,0x6b,0xfb,0x59,0xd0,0xc8,
5183.    0x22,0x36,0xdb,0xd2,0x63,0x98,0x43,0xa1,0x04,0x87,0x86,0xf7,0xa6,0x26,0xbb,0xd6,
5184.    0x59,0x4d,0xbf,0x6a,0x2e,0xaa,0x2b,0xef,0xe6,0x78,0xb6,0x4e,0xe0,0x2f,0xdc,0x7c,
5185.    0xbe,0x57,0x19,0x32,0x7e,0x2a,0xd0,0xb8,0xba,0x29,0x00,0x3c,0x52,0x7d,0xa8,0x49,
5186.    0x3b,0x2d,0xeb,0x25,0x49,0xfa,0xa3,0xaa,0x39,0xa7,0xc5,0xa7,0x50,0x11,0x36,0xfb,
5187.    0xc6,0x67,0x4a,0xf5,0xa5,0x12,0x65,0x7e,0xb0,0xdf,0xaf,0x4e,0xb3,0x61,0x7f,0x2f } };
5188.  unsigned offset=0, entries, tag, type, len, save, c;
5189.  unsigned ver97=0, serial=0, i, wbi=0, wb[4]={0,0,0,0};
5190.  uchar buf97[324], ci, cj, ck;
5191.  short morder, sorder=order;
5192.  char buf[10];
5193./*
```

```
5194.    The MakerNote might have its own TIFF header (possibly with
5195.    its own byte-order!), or it might just be a table.
5196. */
5197.   if (!strcmp(make,"Nokia")) return;
5198.   fread (buf, 1, 10, ifp);
5199.   if (!strncmp (buf,"KDK" ,3) ||        /* these aren't TIFF tables */
5200.       !strncmp (buf,"VER" ,3) ||
5201.       !strncmp (buf,"IIII",4) ||
5202.       !strncmp (buf,"MMMM",4)) return;
5203.   if (!strncmp (buf,"KC"  ,2) ||        /* Konica KD-400Z, KD-510Z */
5204.       !strncmp (buf,"MLY" ,3)) {        /* Minolta DiMAGE G series */
5205.     order = 0x4d4d;
5206.     while ((i=ftell(ifp)) < data_offset && i < 16384) {
5207.       wb[0] = wb[2];  wb[2] = wb[1];  wb[1] = wb[3];
5208.       wb[3] = get2();
5209.       if (wb[1] == 256 && wb[3] == 256 &&
5210.           wb[0] > 256 && wb[0] < 640 && wb[2] > 256 && wb[2] < 640)
5211.         FORC4 cam_mul[c] = wb[c];
5212.     }
5213.     goto quit;
5214.   }
5215.   if (!strcmp (buf,"Nikon")) {
5216.     base = ftell(ifp);
5217.     order = get2();
5218.     if (get2() != 42) goto quit;
5219.     offset = get4();
5220.     fseek (ifp, offset-8, SEEK_CUR);
5221.   } else if (!strcmp (buf,"OLYMPUS") ||
5222.              !strcmp (buf,"PENTAX ")) {
5223.     base = ftell(ifp)-10;
5224.     fseek (ifp, -2, SEEK_CUR);
5225.     order = get2();
5226.     if (buf[0] == 'O') get2();
5227.   } else if (!strncmp (buf,"SONY",4) ||
5228.              !strcmp  (buf,"Panasonic")) {
5229.     goto nf;
5230.   } else if (!strncmp (buf,"FUJIFILM",8)) {
5231.     base = ftell(ifp)-10;
5232. nf: order = 0x4949;
5233.     fseek (ifp,  2, SEEK_CUR);
5234.   } else if (!strcmp (buf,"OLYMP") ||
5235.              !strcmp (buf,"LEICA") ||
5236.              !strcmp (buf,"Ricoh") ||
5237.              !strcmp (buf,"EPSON"))
5238.     fseek (ifp, -2, SEEK_CUR);
5239.   else if (!strcmp (buf,"AOC") ||
5240.            !strcmp (buf,"QVC"))
5241.     fseek (ifp, -4, SEEK_CUR);
5242.   else {
5243.     fseek (ifp, -10, SEEK_CUR);
5244.     if (!strncmp(make,"SAMSUNG",7))
5245.       base = ftell(ifp);
5246.   }
5247.   entries = get2();
5248.   if (entries > 1000) return;
5249.   morder = order;
5250.   while (entries--) {
5251.     order = morder;
5252.     tiff_get (base, &tag, &type, &len, &save);
5253.     tag |= uptag << 16;
5254.     if (tag == 2 && strstr(make,"NIKON") && !iso_speed)
5255.       iso_speed = (get2(),get2());
5256.     if (tag == 4 && len > 26 && len < 35) {
5257.       if ((i=(get4(),get2())) != 0x7fff && !iso_speed)
5258.         iso_speed = 50 * pow (2, i/32.0 - 4);
```

```
5259.       if ((i=(get2(),get2())) != 0x7fff && !aperture)
5260.         aperture = pow (2, i/64.0);
5261.       if ((i=get2()) != 0xffff && !shutter)
5262.         shutter = pow (2, (short) i/-32.0);
5263.       wbi = (get2(),get2());
5264.       shot_order = (get2(),get2());
5265.     }
5266.     if ((tag == 4 || tag == 0x114) && !strncmp(make,"KONICA",6)) {
5267.       fseek (ifp, tag == 4 ? 140:160, SEEK_CUR);
5268.       switch (get2()) {
5269.         case 72:  flip = 0;  break;
5270.         case 76:  flip = 6;  break;
5271.         case 82:  flip = 5;  break;
5272.       }
5273.     }
5274.     if (tag == 7 && type == 2 && len > 20)
5275.       fgets (model2, 64, ifp);
5276.     if (tag == 8 && type == 4)
5277.       shot_order = get4();
5278.     if (tag == 9 && !strcmp(make,"Canon"))
5279.       fread (artist, 64, 1, ifp);
5280.     if (tag == 0xc && len == 4)
5281.       FORC3 cam_mul[(c << 1 | c >> 1) & 3] = getreal(type);
5282.     if (tag == 0xd && type == 7 && get2() == 0xaaaa) {
5283.       for (c=i=2; (ushort) c != 0xbbbb && i < len; i++)
5284.         c = c << 8 | fgetc(ifp);
5285.       while ((i+=4) < len-5)
5286.         if (get4() == 257 && (i=len) && (c = (get4(),fgetc(ifp))) < 3)
5287.           flip = "065"[c]-'0';
5288.     }
5289.     if (tag == 0x10 && type == 4)
5290.       unique_id = get4();
5291.     if (tag == 0x11 && is_raw && !strncmp(make,"NIKON",5)) {
5292.       fseek (ifp, get4()+base, SEEK_SET);
5293.       parse_tiff_ifd (base);
5294.     }
5295.     if (tag == 0x14 && type == 7) {
5296.       if (len == 2560) {
5297.         fseek (ifp, 1248, SEEK_CUR);
5298.         goto get2_256;
5299.       }
5300.       fread (buf, 1, 10, ifp);
5301.       if (!strncmp(buf,"NRW ",4)) {
5302.         fseek (ifp, strcmp(buf+4,"0100") ? 46:1546, SEEK_CUR);
5303.         cam_mul[0] = get4() << 2;
5304.         cam_mul[1] = get4() + get4();
5305.         cam_mul[2] = get4() << 2;
5306.       }
5307.     }
5308.     if (tag == 0x15 && type == 2 && is_raw)
5309.       fread (model, 64, 1, ifp);
5310.     if (strstr(make,"PENTAX")) {
5311.       if (tag == 0x1b) tag = 0x1018;
5312.       if (tag == 0x1c) tag = 0x1017;
5313.     }
5314.     if (tag == 0x1d)
5315.       while ((c = fgetc(ifp)) && c != EOF)
5316.         serial = serial*10 + (isdigit(c) ? c - '0' : c % 10);
5317.     if (tag == 0x29 && type == 1) {
5318.       c = wbi < 18 ? "012347800000005896"[wbi]-'0' : 0;
5319.       fseek (ifp, 8 + c*32, SEEK_CUR);
5320.       FORC4 cam_mul[c ^ (c >> 1) ^ 1] = get4();
5321.     }
5322.     if (tag == 0x3d && type == 3 && len == 4)
5323.       FORC4 cblack[c ^ c >> 1] = get2() >> (14-tiff_bps);
```

```
    if (tag == 0x81 && type == 4) {
      data_offset = get4();
      fseek (ifp, data_offset + 41, SEEK_SET);
      raw_height = get2() * 2;
      raw_width  = get2();
      filters = 0x61616161;
    }
    if ((tag == 0x81  && type == 7) ||
        (tag == 0x100 && type == 7) ||
        (tag == 0x280 && type == 1)) {
      thumb_offset = ftell(ifp);
      thumb_length = len;
    }
    if (tag == 0x88 && type == 4 && (thumb_offset = get4()))
      thumb_offset += base;
    if (tag == 0x89 && type == 4)
      thumb_length = get4();
    if (tag == 0x8c || tag == 0x96)
      meta_offset = ftell(ifp);
    if (tag == 0x97) {
      for (i=0; i < 4; i++)
        ver97 = ver97 * 10 + fgetc(ifp)-'0';
      switch (ver97) {
        case 100:
          fseek (ifp, 68, SEEK_CUR);
          FORC4 cam_mul[(c >> 1) | ((c & 1) << 1)] = get2();
          break;
        case 102:
          fseek (ifp, 6, SEEK_CUR);
          FORC4 cam_mul[c ^ (c >> 1)] = get2();
          break;
        case 103:
          fseek (ifp, 16, SEEK_CUR);
          FORC4 cam_mul[c] = get2();
      }
      if (ver97 >= 200) {
        if (ver97 != 205) fseek (ifp, 280, SEEK_CUR);
        fread (buf97, 324, 1, ifp);
      }
    }
    if (tag == 0xa1 && type == 7) {
      order = 0x4949;
      fseek (ifp, 140, SEEK_CUR);
      FORC3 cam_mul[c] = get4();
    }
    if (tag == 0xa4 && type == 3) {
      fseek (ifp, wbi*48, SEEK_CUR);
      FORC3 cam_mul[c] = get2();
    }
    if (tag == 0xa7 && (unsigned) (ver97-200) < 17) {
      ci = xlat[0][serial & 0xff];
      cj = xlat[1][fgetc(ifp)^fgetc(ifp)^fgetc(ifp)^fgetc(ifp)];
      ck = 0x60;
      for (i=0; i < 324; i++)
        buf97[i] ^= (cj += ci * ck++);
      i = "66666>666;6A;:;55"[ver97-200] - '0';
      FORC4 cam_mul[c ^ (c >> 1) ^ (i & 1)] =
        sget2 (buf97 + (i & -2) + c*2);
    }
    if (tag == 0x200 && len == 3)
      shot_order = (get4(),get4());
    if (tag == 0x200 && len == 4)
      FORC4 cblack[c ^ c >> 1] = get2();
    if (tag == 0x201 && len == 4)
      FORC4 cam_mul[c ^ (c >> 1)] = get2();
```

```
5389.     if (tag == 0x220 && type == 7)
5390.       meta_offset = ftell(ifp);
5391.     if (tag == 0x401 && type == 4 && len == 4)
5392.       FORC4 cblack[c ^ c >> 1] = get4();
5393.     if (tag == 0xe01) {         /* Nikon Capture Note */
5394.       order = 0x4949;
5395.       fseek (ifp, 22, SEEK_CUR);
5396.       for (offset=22; offset+22 < len; offset += 22+i) {
5397.         tag = get4();
5398.         fseek (ifp, 14, SEEK_CUR);
5399.         i = get4()-4;
5400.         if (tag == 0x76a43207) flip = get2();
5401.         else fseek (ifp, i, SEEK_CUR);
5402.       }
5403.     }
5404.     if (tag == 0xe80 && len == 256 && type == 7) {
5405.       fseek (ifp, 48, SEEK_CUR);
5406.       cam_mul[0] = get2() * 508 * 1.078 / 0x10000;
5407.       cam_mul[2] = get2() * 382 * 1.173 / 0x10000;
5408.     }
5409.     if (tag == 0xf00 && type == 7) {
5410.       if (len == 614)
5411.         fseek (ifp, 176, SEEK_CUR);
5412.       else if (len == 734 || len == 1502)
5413.         fseek (ifp, 148, SEEK_CUR);
5414.       else goto next;
5415.       goto get2_256;
5416.     }
5417.     if ((tag == 0x1011 && len == 9) || tag == 0x20400200)
5418.       for (i=0; i < 3; i++)
5419.         FORC3 cmatrix[i][c] = ((short) get2()) / 256.0;
5420.     if ((tag == 0x1012 || tag == 0x20400600) && len == 4)
5421.       FORC4 cblack[c ^ c >> 1] = get2();
5422.     if (tag == 0x1017 || tag == 0x20400100)
5423.       cam_mul[0] = get2() / 256.0;
5424.     if (tag == 0x1018 || tag == 0x20400100)
5425.       cam_mul[2] = get2() / 256.0;
5426.     if (tag == 0x2011 && len == 2) {
5427. get2_256:
5428.       order = 0x4d4d;
5429.       cam_mul[0] = get2() / 256.0;
5430.       cam_mul[2] = get2() / 256.0;
5431.     }
5432.     if ((tag | 0x70) == 0x2070 && (type == 4 || type == 13))
5433.       fseek (ifp, get4()+base, SEEK_SET);
5434.     if (tag == 0x2020 && !strncmp(buf,"OLYMP",5))
5435.       parse_thumb_note (base, 257, 258);
5436.     if (tag == 0x2040)
5437.       parse_makernote (base, 0x2040);
5438.     if (tag == 0xb028) {
5439.       fseek (ifp, get4()+base, SEEK_SET);
5440.       parse_thumb_note (base, 136, 137);
5441.     }
5442.     if (tag == 0x4001 && len > 500) {
5443.       i = len == 582 ? 50 : len == 653 ? 68 : len == 5120 ? 142 : 126;
5444.       fseek (ifp, i, SEEK_CUR);
5445.       FORC4 cam_mul[c ^ (c >> 1)] = get2();
5446.       for (i+=18; i <= len; i+=10) {
5447.         get2();
5448.         FORC4 sraw_mul[c ^ (c >> 1)] = get2();
5449.         if (sraw_mul[1] == 1170) break;
5450.       }
5451.     }
5452.     if (tag == 0x4021 && get4() && get4())
5453.       FORC4 cam_mul[c] = 1024;
```

```
    if (tag == 0xa021)
      FORC4 cam_mul[c ^ (c >> 1)] = get4();
    if (tag == 0xa028)
      FORC4 cam_mul[c ^ (c >> 1)] -= get4();
    if (tag == 0xb001)
      unique_id = get2();
next:
    fseek (ifp, save, SEEK_SET);
  }
quit:
  order = sorder;
}

/*
   Since the TIFF DateTime string has no timezone information,
   assume that the camera's clock was set to Universal Time.
 */
void CLASS get_timestamp (int reversed)
{
  struct tm t;
  char str[20];
  int i;

  str[19] = 0;
  if (reversed)
    for (i=19; i--; ) str[i] = fgetc(ifp);
  else
    fread (str, 19, 1, ifp);
  memset (&t, 0, sizeof t);
  if (sscanf (str, "%d:%d:%d %d:%d:%d", &t.tm_year, &t.tm_mon,
        &t.tm_mday, &t.tm_hour, &t.tm_min, &t.tm_sec) != 6)
    return;
  t.tm_year -= 1900;
  t.tm_mon -= 1;
  t.tm_isdst = -1;
  if (mktime(&t) > 0)
    timestamp = mktime(&t);
}

void CLASS parse_exif (int base)
{
  unsigned kodak, entries, tag, type, len, save, c;
  double expo;

  kodak = !strncmp(make,"EASTMAN",7) && tiff_nifds < 3;
  entries = get2();
  while (entries--) {
    tiff_get (base, &tag, &type, &len, &save);
    switch (tag) {
      case 33434:  tiff_ifd[tiff_nifds-1].shutter =
                   shutter = getreal(type);             break;
      case 33437:  aperture = getreal(type);            break;
      case 34855:  iso_speed = get2();                  break;
      case 36867:
      case 36868:  get_timestamp(0);                    break;
      case 37377:  if ((expo = -getreal(type)) < 128)
                     tiff_ifd[tiff_nifds-1].shutter =
                     shutter = pow (2, expo);           break;
      case 37378:  aperture = pow (2, getreal(type)/2); break;
      case 37386:  focal_len = getreal(type);           break;
      case 37500:  parse_makernote (base, 0);           break;
      case 40962:  if (kodak) raw_width  = get4();      break;
      case 40963:  if (kodak) raw_height = get4();      break;
      case 41730:
        if (get4() == 0x20002)
```

```
5519.           for (exif_cfa=c=0; c < 8; c+=2)
5520.             exif_cfa |= fgetc(ifp) * 0x01010101 << c;
5521.     }
5522.     fseek (ifp, save, SEEK_SET);
5523.   }
5524. }
5525.
5526. void CLASS parse_gps (int base)
5527. {
5528.   unsigned entries, tag, type, len, save, c;
5529.
5530.   entries = get2();
5531.   while (entries--) {
5532.     tiff_get (base, &tag, &type, &len, &save);
5533.     switch (tag) {
5534.       case 1: case 3: case 5:
5535.         gpsdata[29+tag/2] = getc(ifp);                  break;
5536.       case 2: case 4: case 7:
5537.         FORC(6) gpsdata[tag/3*6+c] = get4();            break;
5538.       case 6:
5539.         FORC(2) gpsdata[18+c] = get4();                 break;
5540.       case 18: case 29:
5541.         fgets ((char *) (gpsdata+14+tag/3), MIN(len,12), ifp);
5542.     }
5543.     fseek (ifp, save, SEEK_SET);
5544.   }
5545. }
5546.
5547. void CLASS romm_coeff (float romm_cam[3][3])
5548. {
5549.   static const float rgb_romm[3][3] =   /* ROMM == Kodak ProPhoto */
5550.   { {  2.034193, -0.727420, -0.306766 },
5551.     { -0.228811,  1.231729, -0.002922 },
5552.     { -0.008565, -0.153273,  1.161839 } };
5553.   int i, j, k;
5554.
5555.   for (i=0; i < 3; i++)
5556.     for (j=0; j < 3; j++)
5557.       for (cmatrix[i][j] = k=0; k < 3; k++)
5558.         cmatrix[i][j] += rgb_romm[i][k] * romm_cam[k][j];
5559. }
5560.
5561. void CLASS parse_mos (int offset)
5562. {
5563.   char data[40];
5564.   int skip, from, i, c, neut[4], planes=0, frot=0;
5565.   static const char *mod[] =
5566.   { "","DCB2","Volare","Cantare","CMost","Valeo 6","Valeo 11","Valeo 22",
5567.     "Valeo 11p","Valeo 17","","Aptus 17","Aptus 22","Aptus 75","Aptus 65",
5568.     "Aptus 54S","Aptus 65S","Aptus 75S","AFi 5","AFi 6","AFi 7",
5569.     "AFi-II 7","Aptus-II 7","","Aptus-II 6","","","Aptus-II 10","Aptus-II 5",
5570.     "","","","","Aptus-II 10R","Aptus-II 8","","Aptus-II 12","","AFi-II 12" };
5571.   float romm_cam[3][3];
5572.
5573.   fseek (ifp, offset, SEEK_SET);
5574.   while (1) {
5575.     if (get4() != 0x504b5453) break;
5576.     get4();
5577.     fread (data, 1, 40, ifp);
5578.     skip = get4();
5579.     from = ftell(ifp);
5580.     if (!strcmp(data,"JPEG_preview_data")) {
5581.       thumb_offset = from;
5582.       thumb_length = skip;
5583.     }
```

```
5584.    if (!strcmp(data,"icc_camera_profile")) {
5585.      profile_offset = from;
5586.      profile_length = skip;
5587.    }
5588.    if (!strcmp(data,"ShootObj_back_type")) {
5589.      fscanf (ifp, "%d", &i);
5590.      if ((unsigned) i < sizeof mod / sizeof (*mod))
5591.        strcpy (model, mod[i]);
5592.    }
5593.    if (!strcmp(data,"icc_camera_to_tone_matrix")) {
5594.      for (i=0; i < 9; i++)
5595.        ((float *)romm_cam)[i] = int_to_float(get4());
5596.      romm_coeff (romm_cam);
5597.    }
5598.    if (!strcmp(data,"CaptProf_color_matrix")) {
5599.      for (i=0; i < 9; i++)
5600.        fscanf (ifp, "%f", (float *)romm_cam + i);
5601.      romm_coeff (romm_cam);
5602.    }
5603.    if (!strcmp(data,"CaptProf_number_of_planes"))
5604.      fscanf (ifp, "%d", &planes);
5605.    if (!strcmp(data,"CaptProf_raw_data_rotation"))
5606.      fscanf (ifp, "%d", &flip);
5607.    if (!strcmp(data,"CaptProf_mosaic_pattern"))
5608.      FORC4 {
5609.        fscanf (ifp, "%d", &i);
5610.        if (i == 1) frot = c ^ (c >> 1);
5611.      }
5612.    if (!strcmp(data,"ImgProf_rotation_angle")) {
5613.      fscanf (ifp, "%d", &i);
5614.      flip = i - flip;
5615.    }
5616.    if (!strcmp(data,"NeutObj_neutrals") && !cam_mul[0]) {
5617.      FORC4 fscanf (ifp, "%d", neut+c);
5618.      FORC3 cam_mul[c] = (float) neut[0] / neut[c+1];
5619.    }
5620.    if (!strcmp(data,"Rows_data"))
5621.      load_flags = get4();
5622.    parse_mos (from);
5623.    fseek (ifp, skip+from, SEEK_SET);
5624.  }
5625.  if (planes)
5626.    filters = (planes == 1) * 0x01010101 *
5627.        (uchar) "\x94\x61\x16\x49"[(flip/90 + frot) & 3];
5628.}
5629.
5630.void CLASS linear_table (unsigned len)
5631.{
5632.  int i;
5633.  if (len > 0x1000) len = 0x1000;
5634.  read_shorts (curve, len);
5635.  for (i=len; i < 0x1000; i++)
5636.    curve[i] = curve[i-1];
5637.  maximum = curve[0xfff];
5638.}
5639.
5640.void CLASS parse_kodak_ifd (int base)
5641.{
5642.  unsigned entries, tag, type, len, save;
5643.  int i, c, wbi=-2, wbtemp=6500;
5644.  float mul[3]={1,1,1}, num;
5645.  static const int wbtag[] = { 64037,64040,64039,64041,-1,-1,64042 };
5646.
5647.  entries = get2();
5648.  if (entries > 1024) return;
```

```
  while (entries--) {
    tiff_get (base, &tag, &type, &len, &save);
    if (tag == 1020) wbi = getint(type);
    if (tag == 1021 && len == 72) {             /* WB set in software */
      fseek (ifp, 40, SEEK_CUR);
      FORC3 cam_mul[c] = 2048.0 / get2();
      wbi = -2;
    }
    if (tag == 2118) wbtemp = getint(type);
    if (tag == 2120 + wbi && wbi >= 0)
      FORC3 cam_mul[c] = 2048.0 / getreal(type);
    if (tag == 2130 + wbi)
      FORC3 mul[c] = getreal(type);
    if (tag == 2140 + wbi && wbi >= 0)
      FORC3 {
        for (num=i=0; i < 4; i++)
          num += getreal(type) * pow (wbtemp/100.0, i);
        cam_mul[c] = 2048 / (num * mul[c]);
      }
    if (tag == 2317) linear_table (len);
    if (tag == 6020) iso_speed = getint(type);
    if (tag == 64013) wbi = fgetc(ifp);
    if ((unsigned) wbi < 7 && tag == wbtag[wbi])
      FORC3 cam_mul[c] = get4();
    if (tag == 64019) width = getint(type);
    if (tag == 64020) height = (getint(type)+1) & -2;
    fseek (ifp, save, SEEK_SET);
  }
}

void CLASS parse_minolta (int base);
int CLASS parse_tiff (int base);

int CLASS parse_tiff_ifd (int base)
{
  unsigned entries, tag, type, len, plen=16, save;
  int ifd, use_cm=0, cfa, i, j, c, ima_len=0;
  char software[64], *cbuf, *cp;
  uchar cfa_pat[16], cfa_pc[] = { 0,1,2,3 }, tab[256];
  double cc[4][4], cm[4][3], cam_xyz[4][3], num;
  double ab[]={ 1,1,1,1 }, asn[] = { 0,0,0,0 }, xyz[] = { 1,1,1 };
  unsigned sony_curve[] = { 0,0,0,0,0,4095 };
  unsigned *buf, sony_offset=0, sony_length=0, sony_key=0;
  struct jhead jh;
  FILE *sfp;

  if (tiff_nifds >= sizeof tiff_ifd / sizeof tiff_ifd[0])
    return 1;
  ifd = tiff_nifds++;
  for (j=0; j < 4; j++)
    for (i=0; i < 4; i++)
      cc[j][i] = i == j;
  entries = get2();
  if (entries > 512) return 1;
  while (entries--) {
    tiff_get (base, &tag, &type, &len, &save);
    switch (tag) {
      case 5:   width  = get2();  break;
      case 6:   height = get2();  break;
      case 7:   width += get2();  break;
      case 9:   if ((i = get2())) filters = i;  break;
      case 17: case 18:
        if (type == 3 && len == 1)
          cam_mul[(tag-17)*2] = get2() / 256.0;
        break;
```

```
5714.       case 23:
5715.         if (type == 3) iso_speed = get2();
5716.         break;
5717.       case 28: case 29: case 30:
5718.         cblack[tag-28] = get2();
5719.         cblack[3] = cblack[1];
5720.         break;
5721.       case 36: case 37: case 38:
5722.         cam_mul[tag-36] = get2();
5723.         break;
5724.       case 39:
5725.         if (len < 50 || cam_mul[0]) break;
5726.         fseek (ifp, 12, SEEK_CUR);
5727.         FORC3 cam_mul[c] = get2();
5728.         break;
5729.       case 46:
5730.         if (type != 7 || fgetc(ifp) != 0xff || fgetc(ifp) != 0xd8) break;
5731.         thumb_offset = ftell(ifp) - 2;
5732.         thumb_length = len;
5733.         break;
5734.       case 61440:                       /* Fuji HS10 table */
5735.         fseek (ifp, get4()+base, SEEK_SET);
5736.         parse_tiff_ifd (base);
5737.         break;
5738.       case 2: case 256: case 61441:     /* ImageWidth */
5739.         tiff_ifd[ifd].width = getint(type);
5740.         break;
5741.       case 3: case 257: case 61442:     /* ImageHeight */
5742.         tiff_ifd[ifd].height = getint(type);
5743.         break;
5744.       case 258:                         /* BitsPerSample */
5745.       case 61443:
5746.         tiff_ifd[ifd].samples = len & 7;
5747.         if ((tiff_ifd[ifd].bps = getint(type)) > 32)
5748.             tiff_ifd[ifd].bps = 8;
5749.         if (tiff_bps < tiff_ifd[ifd].bps)
5750.             tiff_bps = tiff_ifd[ifd].bps;
5751.         break;
5752.       case 61446:
5753.         raw_height = 0;
5754.         load_flags = get4() ? 24:80;
5755.         break;
5756.       case 259:                         /* Compression */
5757.         tiff_ifd[ifd].comp = getint(type);
5758.         break;
5759.       case 262:                         /* PhotometricInterpretation */
5760.         tiff_ifd[ifd].phint = get2();
5761.         break;
5762.       case 270:                         /* ImageDescription */
5763.         fread (desc, 512, 1, ifp);
5764.         break;
5765.       case 271:                         /* Make */
5766.         fgets (make, 64, ifp);
5767.         break;
5768.       case 272:                         /* Model */
5769.         fgets (model, 64, ifp);
5770.         break;
5771.       case 280:                         /* Panasonic RW2 offset */
5772.         if (type != 4) break;
5773.         load_raw = &CLASS panasonic_load_raw;
5774.         load_flags = 0x2008;
5775.       case 273:                         /* StripOffset */
5776.       case 513:                         /* JpegIFOffset */
5777.       case 61447:
5778.         tiff_ifd[ifd].offset = get4()+base;
```

```
5779.         if (!tiff_ifd[ifd].bps && tiff_ifd[ifd].offset > 0) {
5780.           fseek (ifp, tiff_ifd[ifd].offset, SEEK_SET);
5781.           if (ljpeg_start (&jh, 1)) {
5782.             tiff_ifd[ifd].comp    = 6;
5783.             tiff_ifd[ifd].width   = jh.wide;
5784.             tiff_ifd[ifd].height  = jh.high;
5785.             tiff_ifd[ifd].bps     = jh.bits;
5786.             tiff_ifd[ifd].samples = jh.clrs;
5787.             if (!(jh.sraw || (jh.clrs & 1)))
5788.               tiff_ifd[ifd].width *= jh.clrs;
5789.             if ((tiff_ifd[ifd].width > 4*tiff_ifd[ifd].height) & ~jh.clrs) {
5790.               tiff_ifd[ifd].width  /= 2;
5791.               tiff_ifd[ifd].height *= 2;
5792.             }
5793.             i = order;
5794.             parse_tiff (tiff_ifd[ifd].offset + 12);
5795.             order = i;
5796.           }
5797.         }
5798.         break;
5799.       case 274:                       /* Orientation */
5800.         tiff_ifd[ifd].flip = "50132467"[get2() & 7]-'0';
5801.         break;
5802.       case 277:                       /* SamplesPerPixel */
5803.         tiff_ifd[ifd].samples = getint(type) & 7;
5804.         break;
5805.       case 279:                       /* StripByteCounts */
5806.       case 514:
5807.       case 61448:
5808.         tiff_ifd[ifd].bytes = get4();
5809.         break;
5810.       case 61454:
5811.         FORC3 cam_mul[(4-c) % 3] = getint(type);
5812.         break;
5813.       case 305:  case 11:             /* Software */
5814.         fgets (software, 64, ifp);
5815.         if (!strncmp(software,"Adobe",5) ||
5816.             !strncmp(software,"dcraw",5) ||
5817.             !strncmp(software,"UFRaw",5) ||
5818.             !strncmp(software,"Bibble",6) ||
5819.             !strncmp(software,"Nikon Scan",10) ||
5820.             !strcmp (software,"Digital Photo Professional"))
5821.           is_raw = 0;
5822.         break;
5823.       case 306:                       /* DateTime */
5824.         get_timestamp(0);
5825.         break;
5826.       case 315:                       /* Artist */
5827.         fread (artist, 64, 1, ifp);
5828.         break;
5829.       case 322:                       /* TileWidth */
5830.         tiff_ifd[ifd].tile_width = getint(type);
5831.         break;
5832.       case 323:                       /* TileLength */
5833.         tiff_ifd[ifd].tile_length = getint(type);
5834.         break;
5835.       case 324:                       /* TileOffsets */
5836.         tiff_ifd[ifd].offset = len > 1 ? ftell(ifp) : get4();
5837.         if (len == 1)
5838.           tiff_ifd[ifd].tile_width = tiff_ifd[ifd].tile_length = 0;
5839.         if (len == 4) {
5840.           load_raw = &CLASS sinar_4shot_load_raw;
5841.           is_raw = 5;
5842.         }
5843.         break;
```

```
5844.       case 330:                           /* SubIFDs */
5845.         if (!strcmp(model,"DSLR-A100") && tiff_ifd[ifd].width == 3872) {
5846.           load_raw = &CLASS sony_arw_load_raw;
5847.           data_offset = get4()+base;
5848.           ifd++;  break;
5849.         }
5850.         while (len--) {
5851.           i = ftell(ifp);
5852.           fseek (ifp, get4()+base, SEEK_SET);
5853.           if (parse_tiff_ifd (base)) break;
5854.           fseek (ifp, i+4, SEEK_SET);
5855.         }
5856.         break;
5857.       case 400:
5858.         strcpy (make, "Sarnoff");
5859.         maximum = 0xfff;
5860.         break;
5861.       case 28688:
5862.         FORC4 sony_curve[c+1] = get2() >> 2 & 0xfff;
5863.         for (i=0; i < 5; i++)
5864.           for (j = sony_curve[i]+1; j <= sony_curve[i+1]; j++)
5865.             curve[j] = curve[j-1] + (1 << i);
5866.         break;
5867.       case 29184: sony_offset = get4();  break;
5868.       case 29185: sony_length = get4();  break;
5869.       case 29217: sony_key    = get4();  break;
5870.       case 29264:
5871.         parse_minolta (ftell(ifp));
5872.         raw_width = 0;
5873.         break;
5874.       case 29443:
5875.         FORC4 cam_mul[c ^ (c < 2)] = get2();
5876.         break;
5877.       case 29459:
5878.         FORC4 cam_mul[c] = get2();
5879.         i = (cam_mul[1] == 1024 && cam_mul[2] == 1024) << 1;
5880.         SWAP (cam_mul[i],cam_mul[i+1])
5881.         break;
5882.       case 33405:                         /* Model2 */
5883.         fgets (model2, 64, ifp);
5884.         break;
5885.       case 33421:                         /* CFARepeatPatternDim */
5886.         if (get2() == 6 && get2() == 6)
5887.           filters = 9;
5888.         break;
5889.       case 33422:                         /* CFAPattern */
5890.         if (filters == 9) {
5891.           FORC(36) ((char *)xtrans)[c] = fgetc(ifp) & 3;
5892.           break;
5893.         }
5894.       case 64777:                         /* Kodak P-series */
5895.         if ((plen=len) > 16) plen = 16;
5896.         fread (cfa_pat, 1, plen, ifp);
5897.         for (colors=cfa=i=0; i < plen && colors < 4; i++) {
5898.           colors += !(cfa & (1 << cfa_pat[i]));
5899.           cfa |= 1 << cfa_pat[i];
5900.         }
5901.         if (cfa == 070) memcpy (cfa_pc,"\003\004\005",3);       /* CMY */
5902.         if (cfa == 072) memcpy (cfa_pc,"\005\003\004\001",4);   /* GMCY */
5903.         goto guess_cfa_pc;
5904.       case 33424:
5905.       case 65024:
5906.         fseek (ifp, get4()+base, SEEK_SET);
5907.         parse_kodak_ifd (base);
5908.         break;
```

```
5909.       case 33434:                         /* ExposureTime */
5910.         tiff_ifd[ifd].shutter = shutter = getreal(type);
5911.         break;
5912.       case 33437:                         /* FNumber */
5913.         aperture = getreal(type);
5914.         break;
5915.       case 34306:                         /* Leaf white balance */
5916.         FORC4 cam_mul[c ^ 1] = 4096.0 / get2();
5917.         break;
5918.       case 34307:                         /* Leaf CatchLight color matrix */
5919.         fread (software, 1, 7, ifp);
5920.         if (strncmp(software,"MATRIX",6)) break;
5921.         colors = 4;
5922.         for (raw_color = i=0; i < 3; i++) {
5923.           FORC4 fscanf (ifp, "%f", &rgb_cam[i][c^1]);
5924.           if (!use_camera_wb) continue;
5925.           num = 0;
5926.           FORC4 num += rgb_cam[i][c];
5927.           FORC4 rgb_cam[i][c] /= num;
5928.         }
5929.         break;
5930.       case 34310:                         /* Leaf metadata */
5931.         parse_mos (ftell(ifp));
5932.       case 34303:
5933.         strcpy (make, "Leaf");
5934.         break;
5935.       case 34665:                         /* EXIF tag */
5936.         fseek (ifp, get4()+base, SEEK_SET);
5937.         parse_exif (base);
5938.         break;
5939.       case 34853:                         /* GPSInfo tag */
5940.         fseek (ifp, get4()+base, SEEK_SET);
5941.         parse_gps (base);
5942.         break;
5943.       case 34675:                         /* InterColorProfile */
5944.       case 50831:                         /* AsShotICCProfile */
5945.         profile_offset = ftell(ifp);
5946.         profile_length = len;
5947.         break;
5948.       case 37122:                         /* CompressedBitsPerPixel */
5949.         kodak_cbpp = get4();
5950.         break;
5951.       case 37386:                         /* FocalLength */
5952.         focal_len = getreal(type);
5953.         break;
5954.       case 37393:                         /* ImageNumber */
5955.         shot_order = getint(type);
5956.         break;
5957.       case 37400:                         /* old Kodak KDC tag */
5958.         for (raw_color = i=0; i < 3; i++) {
5959.           getreal(type);
5960.           FORC3 rgb_cam[i][c] = getreal(type);
5961.         }
5962.         break;
5963.       case 40976:
5964.         strip_offset = get4();
5965.         switch (tiff_ifd[ifd].comp) {
5966.           case 32770: load_raw = &CLASS samsung_load_raw;   break;
5967.           case 32772: load_raw = &CLASS samsung2_load_raw;  break;
5968.           case 32773: load_raw = &CLASS samsung3_load_raw;  break;
5969.         }
5970.         break;
5971.       case 46275:                         /* Imacon tags */
5972.         strcpy (make, "Imacon");
5973.         data_offset = ftell(ifp);
```

```
5974.         ima_len = len;
5975.         break;
5976.       case 46279:
5977.         if (!ima_len) break;
5978.         fseek (ifp, 38, SEEK_CUR);
5979.       case 46274:
5980.         fseek (ifp, 40, SEEK_CUR);
5981.         raw_width  = get4();
5982.         raw_height = get4();
5983.         left_margin = get4() & 7;
5984.         width = raw_width - left_margin - (get4() & 7);
5985.         top_margin = get4() & 7;
5986.         height = raw_height - top_margin - (get4() & 7);
5987.         if (raw_width == 7262) {
5988.           height = 5444;
5989.           width  = 7244;
5990.           left_margin = 7;
5991.         }
5992.         fseek (ifp, 52, SEEK_CUR);
5993.         FORC3 cam_mul[c] = getreal(11);
5994.         fseek (ifp, 114, SEEK_CUR);
5995.         flip = (get2() >> 7) * 90;
5996.         if (width * height * 6 == ima_len) {
5997.           if (flip % 180 == 90) SWAP(width,height);
5998.           raw_width = width;
5999.           raw_height = height;
6000.           left_margin = top_margin = filters = flip = 0;
6001.         }
6002.         sprintf (model, "Ixpress %d-Mp", height*width/1000000);
6003.         load_raw = &CLASS imacon_full_load_raw;
6004.         if (filters) {
6005.           if (left_margin & 1) filters = 0x61616161;
6006.           load_raw = &CLASS unpacked_load_raw;
6007.         }
6008.         maximum = 0xffff;
6009.         break;
6010.       case 50454:                       /* Sinar tag */
6011.       case 50455:
6012.         if (!(cbuf = (char *) malloc(len))) break;
6013.         fread (cbuf, 1, len, ifp);
6014.         for (cp = cbuf-1; cp && cp < cbuf+len; cp = strchr(cp,'\n'))
6015.           if (!strncmp (++cp,"Neutral ",8))
6016.             sscanf (cp+8, "%f %f %f", cam_mul, cam_mul+1, cam_mul+2);
6017.         free (cbuf);
6018.         break;
6019.       case 50458:
6020.         if (!make[0]) strcpy (make, "Hasselblad");
6021.         break;
6022.       case 50459:                       /* Hasselblad tag */
6023.         i = order;
6024.         j = ftell(ifp);
6025.         c = tiff_nifds;
6026.         order = get2();
6027.         fseek (ifp, j+(get2(),get4()), SEEK_SET);
6028.         parse_tiff_ifd (j);
6029.         maximum = 0xffff;
6030.         tiff_nifds = c;
6031.         order = i;
6032.         break;
6033.       case 50706:                       /* DNGVersion */
6034.         FORC4 dng_version = (dng_version << 8) + fgetc(ifp);
6035.         if (!make[0]) strcpy (make, "DNG");
6036.         is_raw = 1;
6037.         break;
6038.       case 50708:                       /* UniqueCameraModel */
```

```
6039.         if (model[0]) break;
6040.         fgets (make, 64, ifp);
6041.         if ((cp = strchr(make,' '))) {
6042.           strcpy(model,cp+1);
6043.           *cp = 0;
6044.         }
6045.         break;
6046.       case 50710:                         /* CFAPlaneColor */
6047.         if (filters == 9) break;
6048.         if (len > 4) len = 4;
6049.         colors = len;
6050.         fread (cfa_pc, 1, colors, ifp);
6051. guess_cfa_pc:
6052.         FORCC tab[cfa_pc[c]] = c;
6053.         cdesc[c] = 0;
6054.         for (i=16; i--; )
6055.           filters = filters << 2 | tab[cfa_pat[i % plen]];
6056.         filters -= !filters;
6057.         break;
6058.       case 50711:                         /* CFALayout */
6059.         if (get2() == 2) fuji_width = 1;
6060.         break;
6061.       case 291:
6062.       case 50712:                         /* LinearizationTable */
6063.         linear_table (len);
6064.         break;
6065.       case 50713:                         /* BlackLevelRepeatDim */
6066.         cblack[4] = get2();
6067.         cblack[5] = get2();
6068.         if (cblack[4] * cblack[5] > sizeof cblack / sizeof *cblack - 6)
6069.             cblack[4] = cblack[5] = 1;
6070.         break;
6071.       case 61450:
6072.         cblack[4] = cblack[5] = MIN(sqrt(len),64);
6073.       case 50714:                         /* BlackLevel */
6074.         if (!(cblack[4] * cblack[5]))
6075.           cblack[4] = cblack[5] = 1;
6076.         FORC (cblack[4] * cblack[5])
6077.           cblack[6+c] = getreal(type);
6078.         black = 0;
6079.         break;
6080.       case 50715:                         /* BlackLevelDeltaH */
6081.       case 50716:                         /* BlackLevelDeltaV */
6082.         for (num=i=0; i < (len & 0xffff); i++)
6083.           num += getreal(type);
6084.         black += num/len + 0.5;
6085.         break;
6086.       case 50717:                         /* WhiteLevel */
6087.         maximum = getint(type);
6088.         break;
6089.       case 50718:                         /* DefaultScale */
6090.         pixel_aspect  = getreal(type);
6091.         pixel_aspect /= getreal(type);
6092.         break;
6093.       case 50721:                         /* ColorMatrix1 */
6094.       case 50722:                         /* ColorMatrix2 */
6095.         FORCC for (j=0; j < 3; j++)
6096.           cm[c][j] = getreal(type);
6097.         use_cm = 1;
6098.         break;
6099.       case 50723:                         /* CameraCalibration1 */
6100.       case 50724:                         /* CameraCalibration2 */
6101.         for (i=0; i < colors; i++)
6102.           FORCC cc[i][c] = getreal(type);
6103.         break;
```

```
6104.       case 50727:                         /* AnalogBalance */
6105.         FORCC ab[c] = getreal(type);
6106.         break;
6107.       case 50728:                         /* AsShotNeutral */
6108.         FORCC asn[c] = getreal(type);
6109.         break;
6110.       case 50729:                         /* AsShotWhiteXY */
6111.         xyz[0] = getreal(type);
6112.         xyz[1] = getreal(type);
6113.         xyz[2] = 1 - xyz[0] - xyz[1];
6114.         FORC3 xyz[c] /= d65_white[c];
6115.         break;
6116.       case 50740:                         /* DNGPrivateData */
6117.         if (dng_version) break;
6118.         parse_minolta (j = get4()+base);
6119.         fseek (ifp, j, SEEK_SET);
6120.         parse_tiff_ifd (base);
6121.         break;
6122.       case 50752:
6123.         read_shorts (cr2_slice, 3);
6124.         break;
6125.       case 50829:                         /* ActiveArea */
6126.         top_margin = getint(type);
6127.         left_margin = getint(type);
6128.         height = getint(type) - top_margin;
6129.         width = getint(type) - left_margin;
6130.         break;
6131.       case 50830:                         /* MaskedAreas */
6132.         for (i=0; i < len && i < 32; i++)
6133.           ((int *)mask)[i] = getint(type);
6134.         black = 0;
6135.         break;
6136.       case 51009:                         /* OpcodeList2 */
6137.         meta_offset = ftell(ifp);
6138.         break;
6139.       case 64772:                         /* Kodak P-series */
6140.         if (len < 13) break;
6141.         fseek (ifp, 16, SEEK_CUR);
6142.         data_offset = get4();
6143.         fseek (ifp, 28, SEEK_CUR);
6144.         data_offset += get4();
6145.         load_raw = &CLASS packed_load_raw;
6146.         break;
6147.       case 65026:
6148.         if (type == 2) fgets (model2, 64, ifp);
6149.     }
6150.     fseek (ifp, save, SEEK_SET);
6151.   }
6152.   if (sony_length && (buf = (unsigned *) malloc(sony_length))) {
6153.     fseek (ifp, sony_offset, SEEK_SET);
6154.     fread (buf, sony_length, 1, ifp);
6155.     sony_decrypt (buf, sony_length/4, 1, sony_key);
6156.     sfp = ifp;
6157.     if ((ifp = tmpfile())) {
6158.       fwrite (buf, sony_length, 1, ifp);
6159.       fseek (ifp, 0, SEEK_SET);
6160.       parse_tiff_ifd (-sony_offset);
6161.       fclose (ifp);
6162.     }
6163.     ifp = sfp;
6164.     free (buf);
6165.   }
6166.   for (i=0; i < colors; i++)
6167.     FORCC cc[i][c] *= ab[i];
6168.   if (use_cm) {
```

```
6169.    FORCC for (i=0; i < 3; i++)
6170.      for (cam_xyz[c][i]=j=0; j < colors; j++)
6171.        cam_xyz[c][i] += cc[c][j] * cm[j][i] * xyz[i];
6172.    cam_xyz_coeff (cmatrix, cam_xyz);
6173.  }
6174.  if (asn[0]) {
6175.    cam_mul[3] = 0;
6176.    FORCC cam_mul[c] = 1 / asn[c];
6177.  }
6178.  if (!use_cm)
6179.    FORCC pre_mul[c] /= cc[c][c];
6180.  return 0;
6181.}
6182.
6183.int CLASS parse_tiff (int base)
6184.{
6185.  int doff;
6186.
6187.  fseek (ifp, base, SEEK_SET);
6188.  order = get2();
6189.  if (order != 0x4949 && order != 0x4d4d) return 0;
6190.  get2();
6191.  while ((doff = get4())) {
6192.    fseek (ifp, doff+base, SEEK_SET);
6193.    if (parse_tiff_ifd (base)) break;
6194.  }
6195.  return 1;
6196.}
6197.
6198.void CLASS apply_tiff()
6199.{
6200.  int max_samp=0, ties=0, os, ns, raw=-1, thm=-1, i;
6201.  struct jhead jh;
6202.
6203.  thumb_misc = 16;
6204.  if (thumb_offset) {
6205.    fseek (ifp, thumb_offset, SEEK_SET);
6206.    if (ljpeg_start (&jh, 1)) {
6207.      thumb_misc   = jh.bits;
6208.      thumb_width  = jh.wide;
6209.      thumb_height = jh.high;
6210.    }
6211.  }
6212.  for (i=tiff_nifds; i--; ) {
6213.    if (tiff_ifd[i].shutter)
6214.      shutter = tiff_ifd[i].shutter;
6215.    tiff_ifd[i].shutter = shutter;
6216.  }
6217.  for (i=0; i < tiff_nifds; i++) {
6218.    if (max_samp < tiff_ifd[i].samples)
6219.        max_samp = tiff_ifd[i].samples;
6220.    if (max_samp > 3) max_samp = 3;
6221.    os = raw_width*raw_height;
6222.    ns = tiff_ifd[i].width*tiff_ifd[i].height;
6223.    if (tiff_bps) {
6224.      os *= tiff_bps;
6225.      ns *= tiff_ifd[i].bps;
6226.    }
6227.    if ((tiff_ifd[i].comp != 6 || tiff_ifd[i].samples != 3) &&
6228.        (tiff_ifd[i].width | tiff_ifd[i].height) < 0x10000 &&
6229.         ns && ((ns > os && (ties = 1)) ||
6230.                (ns == os && shot_select == ties++))) {
6231.      raw_width     = tiff_ifd[i].width;
6232.      raw_height    = tiff_ifd[i].height;
6233.      tiff_bps      = tiff_ifd[i].bps;
```

```
6234.       tiff_compress = tiff_ifd[i].comp;
6235.       data_offset   = tiff_ifd[i].offset;
6236.       tiff_flip     = tiff_ifd[i].flip;
6237.       tiff_samples  = tiff_ifd[i].samples;
6238.       tile_width    = tiff_ifd[i].tile_width;
6239.       tile_length   = tiff_ifd[i].tile_length;
6240.       shutter       = tiff_ifd[i].shutter;
6241.       raw = i;
6242.     }
6243.   }
6244.   if (is_raw == 1 && ties) is_raw = ties;
6245.   if (!tile_width ) tile_width  = INT_MAX;
6246.   if (!tile_length) tile_length = INT_MAX;
6247.   for (i=tiff_nifds; i--; )
6248.     if (tiff_ifd[i].flip) tiff_flip = tiff_ifd[i].flip;
6249.   if (raw >= 0 && !load_raw)
6250.     switch (tiff_compress) {
6251.       case 32767:
6252.         if (tiff_ifd[raw].bytes == raw_width*raw_height) {
6253.           tiff_bps = 12;
6254.           maximum = 4095;
6255.           load_raw = &CLASS sony_arw2_load_raw;                 break;
6256.         }
6257.         if (tiff_ifd[raw].bytes*8 != raw_width*raw_height*tiff_bps) {
6258.           raw_height += 8;
6259.           load_raw = &CLASS sony_arw_load_raw;                  break;
6260.         }
6261.         load_flags = 79;
6262.       case 32769:
6263.         load_flags++;
6264.       case 32770:
6265.       case 32773: goto slr;
6266.       case 0:  case 1:
6267.         if (!strncmp(make,"OLYMPUS",7) &&
6268.                 tiff_ifd[raw].bytes*2 == raw_width*raw_height*3)
6269.           load_flags = 24;
6270.         if (!strcmp(make,"SONY") && tiff_bps < 14 &&
6271.                 tiff_ifd[raw].bytes == raw_width*raw_height*2)
6272.             tiff_bps = 14;
6273.         if (tiff_ifd[raw].bytes*5 == raw_width*raw_height*8) {
6274.           load_flags = 81;
6275.           tiff_bps = 12;
6276.         } slr:
6277.         switch (tiff_bps) {
6278.           case  8: load_raw = &CLASS eight_bit_load_raw;        break;
6279.           case 12: if (tiff_ifd[raw].phint == 2)
6280.                      load_flags = 6;
6281.                    load_raw = &CLASS packed_load_raw;           break;
6282.           case 14: load_raw = &CLASS packed_load_raw;
6283.                    if (tiff_ifd[raw].bytes*4 == raw_width*raw_height*7) break;
6284.                    load_flags = 0;
6285.           case 16: load_raw = &CLASS unpacked_load_raw;
6286.                    if (!strncmp(make,"OLYMPUS",7) &&
6287.                        tiff_ifd[raw].bytes*7 > raw_width*raw_height)
6288.                      load_raw = &CLASS olympus_load_raw;
6289.         }
6290.         if (filters == 9 && tiff_ifd[raw].bytes*8 < raw_width*raw_height*tiff_bps)
6291.           load_raw = &CLASS fuji_xtrans_load_raw;
6292.         break;
6293.       case 6:  case 7:  case 99:
6294.         load_raw = &CLASS lossless_jpeg_load_raw;               break;
6295.       case 262:
6296.         load_raw = &CLASS kodak_262_load_raw;                   break;
6297.       case 34713:
6298.         if ((raw_width+9)/10*16*raw_height == tiff_ifd[raw].bytes) {
```

```
6299.           load_raw = &CLASS packed_load_raw;
6300.           load_flags = 1;
6301.         } else if (raw_width*raw_height*3 == tiff_ifd[raw].bytes*2) {
6302.           load_raw = &CLASS packed_load_raw;
6303.           if (model[0] == 'N') load_flags = 80;
6304.         } else if (raw_width*raw_height*3 == tiff_ifd[raw].bytes) {
6305.           load_raw = &CLASS nikon_yuv_load_raw;
6306.           gamma_curve (1/2.4, 12.92, 1, 4095);
6307.           memset (cblack, 0, sizeof cblack);
6308.           filters = 0;
6309.         } else if (raw_width*raw_height*2 == tiff_ifd[raw].bytes) {
6310.           load_raw = &CLASS unpacked_load_raw;
6311.           load_flags = 4;
6312.           order = 0x4d4d;
6313.         } else
6314.           load_raw = &CLASS nikon_load_raw;                      break;
6315.       case 65535:
6316.         load_raw = &CLASS pentax_load_raw;                       break;
6317.       case 65000:
6318.         switch (tiff_ifd[raw].phint) {
6319.           case 2: load_raw = &CLASS kodak_rgb_load_raw;   filters = 0;  break;
6320.           case 6: load_raw = &CLASS kodak_ycbcr_load_raw; filters = 0;  break;
6321.           case 32803: load_raw = &CLASS kodak_65000_load_raw;
6322.         }
6323.       case 32867: case 34892: break;
6324.       default: is_raw = 0;
6325.     }
6326.   if (!dng_version)
6327.     if ( (tiff_samples == 3 && tiff_ifd[raw].bytes && tiff_bps != 14 &&
6328.           (tiff_compress & -16) != 32768)
6329.       || (tiff_bps == 8 && strncmp(make,"Phase",5) &&
6330.           !strcasestr(make,"Kodak") && !strstr(model2,"DEBUG RAW")))
6331.       is_raw = 0;
6332.   for (i=0; i < tiff_nifds; i++)
6333.     if (i != raw && tiff_ifd[i].samples == max_samp &&
6334.         tiff_ifd[i].width * tiff_ifd[i].height / (SQR(tiff_ifd[i].bps)+1) >
6335.               thumb_width *       thumb_height / (SQR(thumb_misc)+1)
6336.         && tiff_ifd[i].comp != 34892) {
6337.       thumb_width  = tiff_ifd[i].width;
6338.       thumb_height = tiff_ifd[i].height;
6339.       thumb_offset = tiff_ifd[i].offset;
6340.       thumb_length = tiff_ifd[i].bytes;
6341.       thumb_misc   = tiff_ifd[i].bps;
6342.       thm = i;
6343.     }
6344.   if (thm >= 0) {
6345.     thumb_misc |= tiff_ifd[thm].samples << 5;
6346.     switch (tiff_ifd[thm].comp) {
6347.       case 0:
6348.         write_thumb = &CLASS layer_thumb;
6349.         break;
6350.       case 1:
6351.         if (tiff_ifd[thm].bps <= 8)
6352.           write_thumb = &CLASS ppm_thumb;
6353.         else if (!strcmp(make,"Imacon"))
6354.           write_thumb = &CLASS ppm16_thumb;
6355.         else
6356.           thumb_load_raw = &CLASS kodak_thumb_load_raw;
6357.         break;
6358.       case 65000:
6359.         thumb_load_raw = tiff_ifd[thm].phint == 6 ?
6360.                &CLASS kodak_ycbcr_load_raw : &CLASS kodak_rgb_load_raw;
6361.     }
6362.   }
6363. }
```

```
6364.
6365.void CLASS parse_minolta (int base)
6366.{
6367.  int save, tag, len, offset, high=0, wide=0, i, c;
6368.  short sorder=order;
6369.
6370.  fseek (ifp, base, SEEK_SET);
6371.  if (fgetc(ifp) || fgetc(ifp)-'M' || fgetc(ifp)-'R') return;
6372.  order = fgetc(ifp) * 0x101;
6373.  offset = base + get4() + 8;
6374.  while ((save=ftell(ifp)) < offset) {
6375.    for (tag=i=0; i < 4; i++)
6376.      tag = tag << 8 | fgetc(ifp);
6377.    len = get4();
6378.    switch (tag) {
6379.      case 0x505244:                          /* PRD */
6380.        fseek (ifp, 8, SEEK_CUR);
6381.        high = get2();
6382.        wide = get2();
6383.        break;
6384.      case 0x574247:                          /* WBG */
6385.        get4();
6386.        i = strcmp(model,"DiMAGE A200") ? 0:3;
6387.        FORC4 cam_mul[c ^ (c >> 1) ^ i] = get2();
6388.        break;
6389.      case 0x545457:                          /* TTW */
6390.        parse_tiff (ftell(ifp));
6391.        data_offset = offset;
6392.    }
6393.    fseek (ifp, save+len+8, SEEK_SET);
6394.  }
6395.  raw_height = high;
6396.  raw_width  = wide;
6397.  order = sorder;
6398.}
6399.
6400./*
6401.   Many cameras have a "debug mode" that writes JPEG and raw
6402.   at the same time.  The raw file has no header, so try to
6403.   to open the matching JPEG file and read its metadata.
6404. */
6405.void CLASS parse_external_jpeg()
6406.{
6407.  const char *file, *ext;
6408.  char *jname, *jfile, *jext;
6409.  FILE *save=ifp;
6410.
6411.  ext  = strrchr (ifname, '.');
6412.  file = strrchr (ifname, '/');
6413.  if (!file) file = strrchr (ifname, '\\');
6414.  if (!file) file = ifname-1;
6415.  file++;
6416.  if (!ext || strlen(ext) != 4 || ext-file != 8) return;
6417.  jname = (char *) malloc (strlen(ifname) + 1);
6418.  merror (jname, "parse_external_jpeg()");
6419.  strcpy (jname, ifname);
6420.  jfile = file - ifname + jname;
6421.  jext  = ext  - ifname + jname;
6422.  if (strcasecmp (ext, ".jpg")) {
6423.    strcpy (jext, isupper(ext[1]) ? ".JPG":".jpg");
6424.    if (isdigit(*file)) {
6425.      memcpy (jfile, file+4, 4);
6426.      memcpy (jfile+4, file, 4);
6427.    }
6428.  } else
```

```
6429.    while (isdigit(*--jext)) {
6430.      if (*jext != '9') {
6431.        (*jext)++;
6432.        break;
6433.      }
6434.      *jext = '0';
6435.    }
6436.  if (strcmp (jname, ifname)) {
6437.    if ((ifp = fopen (jname, "rb"))) {
6438.      if (verbose)
6439.        fprintf (stderr,_("Reading metadata from %s ...\n"), jname);
6440.      parse_tiff (12);
6441.      thumb_offset = 0;
6442.      is_raw = 1;
6443.      fclose (ifp);
6444.    }
6445.  }
6446.  if (!timestamp)
6447.    fprintf (stderr,_("Failed to read metadata from %s\n"), jname);
6448.  free (jname);
6449.  ifp = save;
6450.}
6451.
6452./*
6453.   CIFF block 0x1030 contains an 8x8 white sample.
6454.   Load this into white[][] for use in scale_colors().
6455. */
6456.void CLASS ciff_block_1030()
6457.{
6458.  static const ushort key[] = { 0x410, 0x45f3 };
6459.  int i, bpp, row, col, vbits=0;
6460.  unsigned long bitbuf=0;
6461.
6462.  if ((get2(),get4()) != 0x80008 || !get4()) return;
6463.  bpp = get2();
6464.  if (bpp != 10 && bpp != 12) return;
6465.  for (i=row=0; row < 8; row++)
6466.    for (col=0; col < 8; col++) {
6467.      if (vbits < bpp) {
6468.        bitbuf = bitbuf << 16 | (get2() ^ key[i++ & 1]);
6469.        vbits += 16;
6470.      }
6471.      white[row][col] = bitbuf >> (vbits -= bpp) & ~(-1 << bpp);
6472.    }
6473.}
6474.
6475./*
6476.   Parse a CIFF file, better known as Canon CRW format.
6477. */
6478.void CLASS parse_ciff (int offset, int length, int depth)
6479.{
6480.  int tboff, nrecs, c, type, len, save, wbi=-1;
6481.  ushort key[] = { 0x410, 0x45f3 };
6482.
6483.  fseek (ifp, offset+length-4, SEEK_SET);
6484.  tboff = get4() + offset;
6485.  fseek (ifp, tboff, SEEK_SET);
6486.  nrecs = get2();
6487.  if ((nrecs | depth) > 127) return;
6488.  while (nrecs--) {
6489.    type = get2();
6490.    len  = get4();
6491.    save = ftell(ifp) + 4;
6492.    fseek (ifp, offset+get4(), SEEK_SET);
6493.    if ((((type >> 8) + 8) | 8) == 0x38)
```

```
6494.       parse_ciff (ftell(ifp), len, depth+1); /* Parse a sub-table */
6495.     if (type == 0x0810)
6496.       fread (artist, 64, 1, ifp);
6497.     if (type == 0x080a) {
6498.       fread (make, 64, 1, ifp);
6499.       fseek (ifp, strlen(make) - 63, SEEK_CUR);
6500.       fread (model, 64, 1, ifp);
6501.     }
6502.     if (type == 0x1810) {
6503.       width = get4();
6504.       height = get4();
6505.       pixel_aspect = int_to_float(get4());
6506.       flip = get4();
6507.     }
6508.     if (type == 0x1835)                 /* Get the decoder table */
6509.       tiff_compress = get4();
6510.     if (type == 0x2007) {
6511.       thumb_offset = ftell(ifp);
6512.       thumb_length = len;
6513.     }
6514.     if (type == 0x1818) {
6515.       shutter = pow (2, -int_to_float((get4(),get4())));
6516.       aperture = pow (2, int_to_float(get4())/2);
6517.     }
6518.     if (type == 0x102a) {
6519.       iso_speed = pow (2, (get4(),get2())/32.0 - 4) * 50;
6520.       aperture  = pow (2, (get2(),(short)get2())/64.0);
6521.       shutter   = pow (2,-((short)get2())/32.0);
6522.       wbi = (get2(),get2());
6523.       if (wbi > 17) wbi = 0;
6524.       fseek (ifp, 32, SEEK_CUR);
6525.       if (shutter > 1e6) shutter = get2()/10.0;
6526.     }
6527.     if (type == 0x102c) {
6528.       if (get2() > 512) {               /* Pro90, G1 */
6529.         fseek (ifp, 118, SEEK_CUR);
6530.         FORC4 cam_mul[c ^ 2] = get2();
6531.       } else {                          /* G2, S30, S40 */
6532.         fseek (ifp, 98, SEEK_CUR);
6533.         FORC4 cam_mul[c ^ (c >> 1) ^ 1] = get2();
6534.       }
6535.     }
6536.     if (type == 0x0032) {
6537.       if (len == 768) {                 /* EOS D30 */
6538.         fseek (ifp, 72, SEEK_CUR);
6539.         FORC4 cam_mul[c ^ (c >> 1)] = 1024.0 / get2();
6540.         if (!wbi) cam_mul[0] = -1;      /* use my auto white balance */
6541.       } else if (!cam_mul[0]) {
6542.         if (get2() == key[0])           /* Pro1, G6, S60, S70 */
6543.           c = (strstr(model,"Pro1") ?
6544.               "012346000000000000":"01345:000000006008")[wbi]-'0'+ 2;
6545.         else {                          /* G3, G5, S45, S50 */
6546.           c = "023457000000006000"[wbi]-'0';
6547.           key[0] = key[1] = 0;
6548.         }
6549.         fseek (ifp, 78 + c*8, SEEK_CUR);
6550.         FORC4 cam_mul[c ^ (c >> 1) ^ 1] = get2() ^ key[c & 1];
6551.         if (!wbi) cam_mul[0] = -1;
6552.       }
6553.     }
6554.     if (type == 0x10a9) {               /* D60, 10D, 300D, and clones */
6555.       if (len > 66) wbi = "0134567028"[wbi]-'0';
6556.       fseek (ifp, 2 + wbi*8, SEEK_CUR);
6557.       FORC4 cam_mul[c ^ (c >> 1)] = get2();
6558.     }
```

```
    if (type == 0x1030 && (0x18040 >> wbi & 1))
      ciff_block_1030();                /* all that don't have 0x10a9 */
    if (type == 0x1031) {
      raw_width = (get2(),get2());
      raw_height = get2();
    }
    if (type == 0x5029) {
      focal_len = len >> 16;
      if ((len & 0xffff) == 2) focal_len /= 32;
    }
    if (type == 0x5813) flash_used = int_to_float(len);
    if (type == 0x5814) canon_ev   = int_to_float(len);
    if (type == 0x5817) shot_order = len;
    if (type == 0x5834) unique_id  = len;
    if (type == 0x580e) timestamp  = len;
    if (type == 0x180e) timestamp  = get4();
#ifdef LOCALTIME
    if ((type | 0x4000) == 0x580e)
      timestamp = mktime (gmtime (&timestamp));
#endif
    fseek (ifp, save, SEEK_SET);
  }
}

void CLASS parse_rollei()
{
  char line[128], *val;
  struct tm t;

  fseek (ifp, 0, SEEK_SET);
  memset (&t, 0, sizeof t);
  do {
    fgets (line, 128, ifp);
    if ((val = strchr(line,'=')))
      *val++ = 0;
    else
      val = line + strlen(line);
    if (!strcmp(line,"DAT"))
      sscanf (val, "%d.%d.%d", &t.tm_mday, &t.tm_mon, &t.tm_year);
    if (!strcmp(line,"TIM"))
      sscanf (val, "%d:%d:%d", &t.tm_hour, &t.tm_min, &t.tm_sec);
    if (!strcmp(line,"HDR"))
      thumb_offset = atoi(val);
    if (!strcmp(line,"X  "))
      raw_width = atoi(val);
    if (!strcmp(line,"Y  "))
      raw_height = atoi(val);
    if (!strcmp(line,"TX "))
      thumb_width = atoi(val);
    if (!strcmp(line,"TY "))
      thumb_height = atoi(val);
  } while (strncmp(line,"EOHD",4));
  data_offset = thumb_offset + thumb_width * thumb_height * 2;
  t.tm_year -= 1900;
  t.tm_mon -= 1;
  if (mktime(&t) > 0)
    timestamp = mktime(&t);
  strcpy (make, "Rollei");
  strcpy (model,"d530flex");
  write_thumb = &CLASS rollei_thumb;
}

void CLASS parse_sinar_ia()
{
  int entries, off;
```

```
6624.  char str[8], *cp;
6625.
6626.  order = 0x4949;
6627.  fseek (ifp, 4, SEEK_SET);
6628.  entries = get4();
6629.  fseek (ifp, get4(), SEEK_SET);
6630.  while (entries--) {
6631.    off = get4(); get4();
6632.    fread (str, 8, 1, ifp);
6633.    if (!strcmp(str,"META"))   meta_offset = off;
6634.    if (!strcmp(str,"THUMB")) thumb_offset = off;
6635.    if (!strcmp(str,"RAW0"))   data_offset = off;
6636.  }
6637.  fseek (ifp, meta_offset+20, SEEK_SET);
6638.  fread (make, 64, 1, ifp);
6639.  make[63] = 0;
6640.  if ((cp = strchr(make,' '))) {
6641.    strcpy (model, cp+1);
6642.    *cp = 0;
6643.  }
6644.  raw_width  = get2();
6645.  raw_height = get2();
6646.  load_raw = &CLASS unpacked_load_raw;
6647.  thumb_width = (get4(),get2());
6648.  thumb_height = get2();
6649.  write_thumb = &CLASS ppm_thumb;
6650.  maximum = 0x3fff;
6651.}
6652.
6653.void CLASS parse_phase_one (int base)
6654.{
6655.  unsigned entries, tag, type, len, data, save, i, c;
6656.  float romm_cam[3][3];
6657.  char *cp;
6658.
6659.  memset (&ph1, 0, sizeof ph1);
6660.  fseek (ifp, base, SEEK_SET);
6661.  order = get4() & 0xffff;
6662.  if (get4() >> 8 != 0x526177) return;          /* "Raw" */
6663.  fseek (ifp, get4()+base, SEEK_SET);
6664.  entries = get4();
6665.  get4();
6666.  while (entries--) {
6667.    tag  = get4();
6668.    type = get4();
6669.    len  = get4();
6670.    data = get4();
6671.    save = ftell(ifp);
6672.    fseek (ifp, base+data, SEEK_SET);
6673.    switch (tag) {
6674.      case 0x100:  flip = "0653"[data & 3]-'0';  break;
6675.      case 0x106:
6676.        for (i=0; i < 9; i++)
6677.          ((float *)romm_cam)[i] = getreal(11);
6678.        romm_coeff (romm_cam);
6679.        break;
6680.      case 0x107:
6681.        FORC3 cam_mul[c] = getreal(11);
6682.        break;
6683.      case 0x108:  raw_width     = data;        break;
6684.      case 0x109:  raw_height    = data;        break;
6685.      case 0x10a:  left_margin   = data;        break;
6686.      case 0x10b:  top_margin    = data;        break;
6687.      case 0x10c:  width         = data;        break;
6688.      case 0x10d:  height        = data;        break;
```

```
      case 0x10e:  ph1.format    = data;        break;
      case 0x10f:  data_offset   = data+base;   break;
      case 0x110:  meta_offset   = data+base;
                   meta_length   = len;                 break;
      case 0x112:  ph1.key_off   = save - 4;            break;
      case 0x210:  ph1.tag_210   = int_to_float(data);  break;
      case 0x21a:  ph1.tag_21a   = data;                break;
      case 0x21c:  strip_offset  = data+base;           break;
      case 0x21d:  ph1.black     = data;                break;
      case 0x222:  ph1.split_col = data;                break;
      case 0x223:  ph1.black_col = data+base;           break;
      case 0x224:  ph1.split_row = data;                break;
      case 0x225:  ph1.black_row = data+base;           break;
      case 0x301:
        model[63] = 0;
        fread (model, 1, 63, ifp);
        if ((cp = strstr(model," camera"))) *cp = 0;
    }
    fseek (ifp, save, SEEK_SET);
  }
  load_raw = ph1.format < 3 ?
        &CLASS phase_one_load_raw : &CLASS phase_one_load_raw_c;
  maximum = 0xffff;
  strcpy (make, "Phase One");
  if (model[0]) return;
  switch (raw_height) {
    case 2060: strcpy (model,"LightPhase");     break;
    case 2682: strcpy (model,"H 10");           break;
    case 4128: strcpy (model,"H 20");           break;
    case 5488: strcpy (model,"H 25");           break;
  }
}

void CLASS parse_fuji (int offset)
{
  unsigned entries, tag, len, save, c;

  fseek (ifp, offset, SEEK_SET);
  entries = get4();
  if (entries > 255) return;
  while (entries--) {
    tag = get2();
    len = get2();
    save = ftell(ifp);
    if (tag == 0x100) {
      raw_height = get2();
      raw_width  = get2();
    } else if (tag == 0x121) {
      height = get2();
      if ((width = get2()) == 4284) width += 3;
    } else if (tag == 0x130) {
      fuji_layout = fgetc(ifp) >> 7;
      fuji_width = !(fgetc(ifp) & 8);
    } else if (tag == 0x131) {
      filters = 9;
      FORC(36) xtrans_abs[0][35-c] = fgetc(ifp) & 3;
    } else if (tag == 0x2ff0) {
      FORC4 cam_mul[c ^ 1] = get2();
    } else if (tag == 0xc000 && len > 20000) {
      c = order;
      order = 0x4949;
      while ((tag = get4()) > raw_width);
      width = tag;
      height = get4();
      order = c;
```

```
6754.    }
6755.    fseek (ifp, save+len, SEEK_SET);
6756.  }
6757.  height <<= fuji_layout;
6758.  width  >>= fuji_layout;
6759.}
6760.
6761.int CLASS parse_jpeg (int offset)
6762.{
6763.  int len, save, hlen, mark;
6764.
6765.  fseek (ifp, offset, SEEK_SET);
6766.  if (fgetc(ifp) != 0xff || fgetc(ifp) != 0xd8) return 0;
6767.
6768.  while (fgetc(ifp) == 0xff && (mark = fgetc(ifp)) != 0xda) {
6769.    order = 0x4d4d;
6770.    len   = get2() - 2;
6771.    save  = ftell(ifp);
6772.    if (mark == 0xc0 || mark == 0xc3 || mark == 0xc9) {
6773.      fgetc(ifp);
6774.      raw_height = get2();
6775.      raw_width  = get2();
6776.    }
6777.    order = get2();
6778.    hlen  = get4();
6779.    if (get4() == 0x48454150)           /* "HEAP" */
6780.      parse_ciff (save+hlen, len-hlen, 0);
6781.    if (parse_tiff (save+6)) apply_tiff();
6782.    fseek (ifp, save+len, SEEK_SET);
6783.  }
6784.  return 1;
6785.}
6786.
6787.void CLASS parse_riff()
6788.{
6789.  unsigned i, size, end;
6790.  char tag[4], date[64], month[64];
6791.  static const char mon[12][4] =
6792.  { "Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec" };
6793.  struct tm t;
6794.
6795.  order = 0x4949;
6796.  fread (tag, 4, 1, ifp);
6797.  size = get4();
6798.  end = ftell(ifp) + size;
6799.  if (!memcmp(tag,"RIFF",4) || !memcmp(tag,"LIST",4)) {
6800.    get4();
6801.    while (ftell(ifp)+7 < end && !feof(ifp))
6802.      parse_riff();
6803.  } else if (!memcmp(tag,"nctg",4)) {
6804.    while (ftell(ifp)+7 < end) {
6805.      i = get2();
6806.      size = get2();
6807.      if ((i+1) >> 1 == 10 && size == 20)
6808.        get_timestamp(0);
6809.      else fseek (ifp, size, SEEK_CUR);
6810.    }
6811.  } else if (!memcmp(tag,"IDIT",4) && size < 64) {
6812.    fread (date, 64, 1, ifp);
6813.    date[size] = 0;
6814.    memset (&t, 0, sizeof t);
6815.    if (sscanf (date, "%*s %s %d %d:%d:%d %d", month, &t.tm_mday,
6816.        &t.tm_hour, &t.tm_min, &t.tm_sec, &t.tm_year) == 6) {
6817.      for (i=0; i < 12 && strcasecmp(mon[i],month); i++);
6818.      t.tm_mon = i;
```

```
6819.      t.tm_year -= 1900;
6820.      if (mktime(&t) > 0)
6821.        timestamp = mktime(&t);
6822.    }
6823.  } else
6824.    fseek (ifp, size, SEEK_CUR);
6825.}
6826.
6827.void CLASS parse_crx (int end)
6828.{
6829.  unsigned i, save, size, tag, base;
6830.  static int index=0, wide, high, off, len;
6831.
6832.  order = 0x4d4d;
6833.  while (ftell(ifp)+7 < end) {
6834.    save = ftell(ifp);
6835.    if ((size = get4()) < 8) break;
6836.    switch (tag = get4()) {
6837.      case 0x6d6f6f76:                          /* moov */
6838.      case 0x7472616b:                          /* trak */
6839.      case 0x6d646961:                          /* mdia */
6840.      case 0x6d696e66:                          /* minf */
6841.      case 0x7374626c:                          /* stbl */
6842.        parse_crx (save+size);
6843.        break;
6844.      case 0x75756964:                          /* uuid */
6845.        switch (i=get4()) {
6846.          case 0xeaf42b5e: fseek (ifp,  8, SEEK_CUR);
6847.          case 0x85c0b687: fseek (ifp, 12, SEEK_CUR);
6848.            parse_crx (save+size);
6849.        }
6850.        break;
6851.      case 0x434d5431:                          /* CMT1 */
6852.      case 0x434d5432:                          /* CMT2 */
6853.        base = ftell(ifp);
6854.        order = get2();
6855.        fseek (ifp, 6, SEEK_CUR);
6856.        tag & 1 ? parse_tiff_ifd (base) : parse_exif (base);
6857.        order = 0x4d4d;
6858.        break;
6859.      case 0x746b6864:                          /* tkhd */
6860.        fseek (ifp, 12, SEEK_CUR);
6861.        index = get4();
6862.        fseek (ifp, 58, SEEK_CUR);
6863.        wide = get4();
6864.        high = get4();
6865.        break;
6866.      case 0x7374737a:                          /* stsz */
6867.        len = (get4(),get4());
6868.        break;
6869.      case 0x636f3634:                          /* co64 */
6870.        fseek (ifp, 12, SEEK_CUR);
6871.        off = get4();
6872.        switch (index) {
6873.          case 1:                       /* 1 = full size, 2 = 27% size */
6874.            thumb_width  = wide;
6875.            thumb_height = high;
6876.            thumb_length = len;
6877.            thumb_offset = off;
6878.            break;
6879.          case 3:
6880.            raw_width  = wide;
6881.            raw_height = high;
6882.            data_offset = off;
6883.            load_raw = &CLASS canon_crx_load_raw;
```

```
6884.        }
6885.        break;
6886.      case 0x50525657:                          /* PRVW */
6887.        fseek (ifp, 6, SEEK_CUR);
6888.    }
6889.    fseek (ifp, save+size, SEEK_SET);
6890.  }
6891.}
6892.
6893.void CLASS parse_qt (int end)
6894.{
6895.  unsigned save, size;
6896.  char tag[4];
6897.
6898.  order = 0x4d4d;
6899.  while (ftell(ifp)+7 < end) {
6900.    save = ftell(ifp);
6901.    if ((size = get4()) < 8) return;
6902.    fread (tag, 4, 1, ifp);
6903.    if (!memcmp(tag,"moov",4) ||
6904.        !memcmp(tag,"udta",4) ||
6905.        !memcmp(tag,"CNTH",4))
6906.      parse_qt (save+size);
6907.    if (!memcmp(tag,"CNDA",4))
6908.      parse_jpeg (ftell(ifp));
6909.    fseek (ifp, save+size, SEEK_SET);
6910.  }
6911.}
6912.
6913.void CLASS parse_smal (int offset, int fsize)
6914.{
6915.  int ver;
6916.
6917.  fseek (ifp, offset+2, SEEK_SET);
6918.  order = 0x4949;
6919.  ver = fgetc(ifp);
6920.  if (ver == 6)
6921.    fseek (ifp, 5, SEEK_CUR);
6922.  if (get4() != fsize) return;
6923.  if (ver > 6) data_offset = get4();
6924.  raw_height = height = get2();
6925.  raw_width  = width  = get2();
6926.  strcpy (make, "SMaL");
6927.  sprintf (model, "v%d %dx%d", ver, width, height);
6928.  if (ver == 6) load_raw = &CLASS smal_v6_load_raw;
6929.  if (ver == 9) load_raw = &CLASS smal_v9_load_raw;
6930.}
6931.
6932.void CLASS parse_cine()
6933.{
6934.  unsigned off_head, off_setup, off_image, i;
6935.
6936.  order = 0x4949;
6937.  fseek (ifp, 4, SEEK_SET);
6938.  is_raw = get2() == 2;
6939.  fseek (ifp, 14, SEEK_CUR);
6940.  is_raw *= get4();
6941.  off_head = get4();
6942.  off_setup = get4();
6943.  off_image = get4();
6944.  timestamp = get4();
6945.  if ((i = get4())) timestamp = i;
6946.  fseek (ifp, off_head+4, SEEK_SET);
6947.  raw_width = get4();
6948.  raw_height = get4();
```

```
  switch (get2(),get2()) {
    case  8:  load_raw = &CLASS eight_bit_load_raw;  break;
    case 16:  load_raw = &CLASS  unpacked_load_raw;
  }
  fseek (ifp, off_setup+792, SEEK_SET);
  strcpy (make, "CINE");
  sprintf (model, "%d", get4());
  fseek (ifp, 12, SEEK_CUR);
  switch ((i=get4()) & 0xffffff) {
    case  3:  filters = 0x94949494;  break;
    case  4:  filters = 0x49494949;  break;
    default:  is_raw = 0;
  }
  fseek (ifp, 72, SEEK_CUR);
  switch ((get4()+3600) % 360) {
    case 270:  flip = 4;  break;
    case 180:  flip = 1;  break;
    case  90:  flip = 7;  break;
    case   0:  flip = 2;
  }
  cam_mul[0] = getreal(11);
  cam_mul[2] = getreal(11);
  maximum = ~(-1 << get4());
  fseek (ifp, 668, SEEK_CUR);
  shutter = get4()/1000000000.0;
  fseek (ifp, off_image, SEEK_SET);
  if (shot_select < is_raw)
    fseek (ifp, shot_select*8, SEEK_CUR);
  data_offset  = (INT64) get4() + 8;
  data_offset += (INT64) get4() << 32;
}

void CLASS parse_redcine()
{
  unsigned i, len, rdvo;

  order = 0x4d4d;
  is_raw = 0;
  fseek (ifp, 52, SEEK_SET);
  width  = get4();
  height = get4();
  fseek (ifp, 0, SEEK_END);
  fseek (ifp, -(i = ftello(ifp) & 511), SEEK_CUR);
  if (get4() != i || get4() != 0x52454f42) {
    fprintf (stderr,_("%s: Tail is missing, parsing from head...\n"), ifname);
    fseek (ifp, 0, SEEK_SET);
    while ((len = get4()) != EOF) {
      if (get4() == 0x52454456)
        if (is_raw++ == shot_select)
          data_offset = ftello(ifp) - 8;
      fseek (ifp, len-8, SEEK_CUR);
    }
  } else {
    rdvo = get4();
    fseek (ifp, 12, SEEK_CUR);
    is_raw = get4();
    fseeko (ifp, rdvo+8 + shot_select*4, SEEK_SET);
    data_offset = get4();
  }
}

char * CLASS foveon_gets (int offset, char *str, int len)
{
  int i;
  fseek (ifp, offset, SEEK_SET);
```

```
  for (i=0; i < len-1; i++)
    if ((str[i] = get2()) == 0) break;
  str[i] = 0;
  return str;
}

void CLASS parse_foveon()
{
  int entries, img=0, off, len, tag, save, i, wide, high, pent, poff[256][2];
  char name[64], value[64];

  order = 0x4949;                       /* Little-endian */
  fseek (ifp, 36, SEEK_SET);
  flip = get4();
  fseek (ifp, -4, SEEK_END);
  fseek (ifp, get4(), SEEK_SET);
  if (get4() != 0x64434553) return;     /* SECd */
  entries = (get4(),get4());
  while (entries--) {
    off = get4();
    len = get4();
    tag = get4();
    save = ftell(ifp);
    fseek (ifp, off, SEEK_SET);
    if (get4() != (0x20434553 | (tag << 24))) return;
    switch (tag) {
      case 0x47414d49:                  /* IMAG */
      case 0x32414d49:                  /* IMA2 */
        fseek (ifp, 8, SEEK_CUR);
        pent = get4();
        wide = get4();
        high = get4();
        if (wide > raw_width && high > raw_height) {
          switch (pent) {
            case  5:  load_flags = 1;
            case  6:  load_raw = &CLASS foveon_sd_load_raw;  break;
            case 30:  load_raw = &CLASS foveon_dp_load_raw;  break;
            default:  load_raw = 0;
          }
          raw_width  = wide;
          raw_height = high;
          data_offset = off+28;
          is_foveon = 1;
        }
        fseek (ifp, off+28, SEEK_SET);
        if (fgetc(ifp) == 0xff && fgetc(ifp) == 0xd8
                && thumb_length < len-28) {
          thumb_offset = off+28;
          thumb_length = len-28;
          write_thumb = &CLASS jpeg_thumb;
        }
        if (++img == 2 && !thumb_length) {
          thumb_offset = off+24;
          thumb_width = wide;
          thumb_height = high;
          write_thumb = &CLASS foveon_thumb;
        }
        break;
      case 0x464d4143:                  /* CAMF */
        meta_offset = off+8;
        meta_length = len-28;
        break;
      case 0x504f5250:                  /* PROP */
        pent = (get4(),get4());
        fseek (ifp, 12, SEEK_CUR);
```

```
7079.         off += pent*8 + 24;
7080.         if ((unsigned) pent > 256) pent=256;
7081.         for (i=0; i < pent*2; i++)
7082.           ((int *)poff)[i] = off + get4()*2;
7083.         for (i=0; i < pent; i++) {
7084.           foveon_gets (poff[i][0], name, 64);
7085.           foveon_gets (poff[i][1], value, 64);
7086.           if (!strcmp (name, "ISO"))
7087.             iso_speed = atoi(value);
7088.           if (!strcmp (name, "CAMMANUF"))
7089.             strcpy (make, value);
7090.           if (!strcmp (name, "CAMMODEL"))
7091.             strcpy (model, value);
7092.           if (!strcmp (name, "WB_DESC"))
7093.             strcpy (model2, value);
7094.           if (!strcmp (name, "TIME"))
7095.             timestamp = atoi(value);
7096.           if (!strcmp (name, "EXPTIME"))
7097.             shutter = atoi(value) / 1000000.0;
7098.           if (!strcmp (name, "APERTURE"))
7099.             aperture = atof(value);
7100.           if (!strcmp (name, "FLENGTH"))
7101.             focal_len = atof(value);
7102.         }
7103. #ifdef LOCALTIME
7104.         timestamp = mktime (gmtime (&timestamp));
7105. #endif
7106.     }
7107.     fseek (ifp, save, SEEK_SET);
7108.   }
7109. }
7110.
7111. /*
7112.    All matrices are from Adobe DNG Converter unless otherwise noted.
7113.  */
7114. void CLASS adobe_coeff (const char *make, const char *model)
7115. {
7116.   static const struct {
7117.     const char *prefix;
7118.     short black, maximum, trans[12];
7119.   } table[] = {
7120.     { "AgfaPhoto DC-833m", 0, 0,        /* DJC */
7121.         { 11438,-3762,-1115,-2409,9914,2497,-1227,2295,5300 } },
7122.     { "Apple QuickTake", 0, 0,          /* DJC */
7123.         { 21392,-5653,-3353,2406,8010,-415,7166,1427,2078 } },
7124.     { "Canon EOS D2000", 0, 0,
7125.         { 24542,-10860,-3401,-1490,11370,-297,2858,-605,3225 } },
7126.     { "Canon EOS D6000", 0, 0,
7127.         { 20482,-7172,-3125,-1033,10410,-285,2542,226,3136 } },
7128.     { "Canon EOS D30", 0, 0,
7129.         { 9805,-2689,-1312,-5803,13064,3068,-2438,3075,8775 } },
7130.     { "Canon EOS D60", 0, 0xfa0,
7131.         { 6188,-1341,-890,-7168,14489,2937,-2640,3228,8483 } },
7132.     { "Canon EOS 5DS", 0, 0x3c96,
7133.         { 6250,-711,-808,-5153,12794,2636,-1249,2198,5610 } },
7134.     { "Canon EOS 5D Mark IV", 0, 0,
7135.         { 6446,-366,-864,-4436,12204,2513,-952,2496,6348 } },
7136.     { "Canon EOS 5D Mark III", 0, 0x3c80,
7137.         { 6722,-635,-963,-4287,12460,2028,-908,2162,5668 } },
7138.     { "Canon EOS 5D Mark II", 0, 0x3cf0,
7139.         { 4716,603,-830,-7798,15474,2480,-1496,1937,6651 } },
7140.     { "Canon EOS 5D", 0, 0xe6c,
7141.         { 6347,-479,-972,-8297,15954,2480,-1968,2131,7649 } },
7142.     { "Canon EOS 6D Mark II", 0, 0,
7143.         { 6875,-970,-932,-4691,12459,2501,-874,1953,5809 } },
```

```
7144.     { "Canon EOS 6D", 0, 0x3c82,
7145.         { 7034,-804,-1014,-4420,12564,2058,-851,1994,5758 } },
7146.     { "Canon EOS 7D Mark II", 0, 0x3510,
7147.         { 7268,-1082,-969,-4186,11839,2663,-825,2029,5839 } },
7148.     { "Canon EOS 7D", 0, 0x3510,
7149.         { 6844,-996,-856,-3876,11761,2396,-593,1772,6198 } },
7150.     { "Canon EOS 10D", 0, 0xfa0,
7151.         { 8197,-2000,-1118,-6714,14335,2592,-2536,3178,8266 } },
7152.     { "Canon EOS 20Da", 0, 0,
7153.         { 14155,-5065,-1382,-6550,14633,2039,-1623,1824,6561 } },
7154.     { "Canon EOS 20D", 0, 0xfff,
7155.         { 6599,-537,-891,-8071,15783,2424,-1983,2234,7462 } },
7156.     { "Canon EOS 30D", 0, 0,
7157.         { 6257,-303,-1000,-7880,15621,2396,-1714,1904,7046 } },
7158.     { "Canon EOS 40D", 0, 0x3f60,
7159.         { 6071,-747,-856,-7653,15365,2441,-2025,2553,7315 } },
7160.     { "Canon EOS 50D", 0, 0x3d93,
7161.         { 4920,616,-593,-6493,13964,2784,-1774,3178,7005 } },
7162.     { "Canon EOS 60D", 0, 0x2ff7,
7163.         { 6719,-994,-925,-4408,12426,2211,-887,2129,6051 } },
7164.     { "Canon EOS 70D", 0, 0x3bc7,
7165.         { 7034,-804,-1014,-4420,12564,2058,-851,1994,5758 } },
7166.     { "Canon EOS 77D", 0, 0,
7167.         { 7377,-742,-998,-4235,11981,2549,-673,1918,5538 } },
7168.     { "Canon EOS 80D", 0, 0,
7169.         { 7457,-671,-937,-4849,12495,2643,-1213,2354,5492 } },
7170.     { "Canon EOS 100D", 0, 0x350f,
7171.         { 6602,-841,-939,-4472,12458,2247,-975,2039,6148 } },
7172.     { "Canon EOS 200D", 0, 0,
7173.         { 7377,-742,-998,-4235,11981,2549,-673,1918,5538 } },
7174.     { "Canon EOS 300D", 0, 0xfa0,
7175.         { 8197,-2000,-1118,-6714,14335,2592,-2536,3178,8266 } },
7176.     { "Canon EOS 350D", 0, 0xfff,
7177.         { 6018,-617,-965,-8645,15881,2975,-1530,1719,7642 } },
7178.     { "Canon EOS 400D", 0, 0xe8e,
7179.         { 7054,-1501,-990,-8156,15544,2812,-1278,1414,7796 } },
7180.     { "Canon EOS 450D", 0, 0x390d,
7181.         { 5784,-262,-821,-7539,15064,2672,-1982,2681,7427 } },
7182.     { "Canon EOS 500D", 0, 0x3479,
7183.         { 4763,712,-646,-6821,14399,2640,-1921,3276,6561 } },
7184.     { "Canon EOS 550D", 0, 0x3dd7,
7185.         { 6941,-1164,-857,-3825,11597,2534,-416,1540,6039 } },
7186.     { "Canon EOS 600D", 0, 0x3510,
7187.         { 6461,-907,-882,-4300,12184,2378,-819,1944,5931 } },
7188.     { "Canon EOS 650D", 0, 0x354d,
7189.         { 6602,-841,-939,-4472,12458,2247,-975,2039,6148 } },
7190.     { "Canon EOS 700D", 0, 0x3c00,
7191.         { 6602,-841,-939,-4472,12458,2247,-975,2039,6148 } },
7192.     { "Canon EOS 750D", 0, 0x368e,
7193.         { 6362,-823,-847,-4426,12109,2616,-743,1857,5635 } },
7194.     { "Canon EOS 760D", 0, 0x350f,
7195.         { 6362,-823,-847,-4426,12109,2616,-743,1857,5635 } },
7196.     { "Canon EOS 800D", 0, 0,
7197.         { 6970,-512,-968,-4425,12161,2553,-739,1982,5601 } },
7198.     { "Canon EOS 1000D", 0, 0xe43,
7199.         { 6771,-1139,-977,-7818,15123,2928,-1244,1437,7533 } },
7200.     { "Canon EOS 1100D", 0, 0x3510,
7201.         { 6444,-904,-893,-4563,12308,2535,-903,2016,6728 } },
7202.     { "Canon EOS 1200D", 0, 0x37c2,
7203.         { 6461,-907,-882,-4300,12184,2378,-819,1944,5931 } },
7204.     { "Canon EOS 1300D", 0, 0x3510,
7205.         { 6939,-1016,-866,-4428,12473,2177,-1175,2178,6162 } },
7206.     { "Canon EOS 1500D", 0, 0,
7207.         { 8532,-701,-1167,-4095,11879,2508,-797,2424,7010 } },
7208.     { "Canon EOS 3000D", 0, 0,
```

```
7209.        { 6939,-1016,-866,-4428,12473,2177,-1175,2178,6162 } },
7210.    { "Canon EOS M6", 0, 0,
7211.        { 8532,-701,-1167,-4095,11879,2508,-797,2424,7010 } },
7212.    { "Canon EOS M5", 0, 0,     /* also M50 */
7213.        { 8532,-701,-1167,-4095,11879,2508,-797,2424,7010 } },
7214.    { "Canon EOS M3", 0, 0,
7215.        { 6362,-823,-847,-4426,12109,2616,-743,1857,5635 } },
7216.    { "Canon EOS M100", 0, 0,
7217.        { 8532,-701,-1167,-4095,11879,2508,-797,2424,7010 } },
7218.    { "Canon EOS M10", 0, 0,
7219.        { 6400,-480,-888,-5294,13416,2047,-1296,2203,6137 } },
7220.    { "Canon EOS M", 0, 0,
7221.        { 6602,-841,-939,-4472,12458,2247,-975,2039,6148 } },
7222.    { "Canon EOS-1Ds Mark III", 0, 0x3bb0,
7223.        { 5859,-211,-930,-8255,16017,2353,-1732,1887,7448 } },
7224.    { "Canon EOS-1Ds Mark II", 0, 0xe80,
7225.        { 6517,-602,-867,-8180,15926,2378,-1618,1771,7633 } },
7226.    { "Canon EOS-1D Mark IV", 0, 0x3bb0,
7227.        { 6014,-220,-795,-4109,12014,2361,-561,1824,5787 } },
7228.    { "Canon EOS-1D Mark III", 0, 0x3bb0,
7229.        { 6291,-540,-976,-8350,16145,2311,-1714,1858,7326 } },
7230.    { "Canon EOS-1D Mark II N", 0, 0xe80,
7231.        { 6240,-466,-822,-8180,15825,2500,-1801,1938,8042 } },
7232.    { "Canon EOS-1D Mark II", 0, 0xe80,
7233.        { 6264,-582,-724,-8312,15948,2504,-1744,1919,8664 } },
7234.    { "Canon EOS-1DS", 0, 0xe20,
7235.        { 4374,3631,-1743,-7520,15212,2472,-2892,3632,8161 } },
7236.    { "Canon EOS-1D C", 0, 0x3c4e,
7237.        { 6847,-614,-1014,-4669,12737,2139,-1197,2488,6846 } },
7238.    { "Canon EOS-1D X Mark II", 0, 0,
7239.        { 7596,-978,-967,-4808,12571,2503,-1398,2567,5752 } },
7240.    { "Canon EOS-1D X", 0, 0x3c4e,
7241.        { 6847,-614,-1014,-4669,12737,2139,-1197,2488,6846 } },
7242.    { "Canon EOS-1D", 0, 0xe20,
7243.        { 6806,-179,-1020,-8097,16415,1687,-3267,4236,7690 } },
7244.    { "Canon EOS C500", 853, 0,         /* DJC */
7245.        { 17851,-10604,922,-7425,16662,763,-3660,3636,22278 } },
7246.    { "Canon PowerShot A530", 0, 0,
7247.        { 0 } },        /* don't want the A5 matrix */
7248.    { "Canon PowerShot A50", 0, 0,
7249.        { -5300,9846,1776,3436,684,3939,-5540,9879,6200,-1404,11175,217 } },
7250.    { "Canon PowerShot A5", 0, 0,
7251.        { -4801,9475,1952,2926,1611,4094,-5259,10164,5947,-1554,10883,547 } },
7252.    { "Canon PowerShot G10", 0, 0,
7253.        { 11093,-3906,-1028,-5047,12492,2879,-1003,1750,5561 } },
7254.    { "Canon PowerShot G11", 0, 0,
7255.        { 12177,-4817,-1069,-1612,9864,2049,-98,850,4471 } },
7256.    { "Canon PowerShot G12", 0, 0,
7257.        { 13244,-5501,-1248,-1508,9858,1935,-270,1083,4366 } },
7258.    { "Canon PowerShot G15", 0, 0,
7259.        { 7474,-2301,-567,-4056,11456,2975,-222,716,4181 } },
7260.    { "Canon PowerShot G16", 0, 0,
7261.        { 8020,-2687,-682,-3704,11879,2052,-965,1921,5556 } },
7262.    { "Canon PowerShot G1 X Mark III", 0, 0,
7263.        { 8532,-701,-1167,-4095,11879,2508,-797,2424,7010 } },
7264.    { "Canon PowerShot G1 X", 0, 0,
7265.        { 7378,-1255,-1043,-4088,12251,2048,-876,1946,5805 } },
7266.    { "Canon PowerShot G1", 0, 0,
7267.        { -4778,9467,2172,4743,-1141,4344,-5146,9908,6077,-1566,11051,557 } },
7268.    { "Canon PowerShot G2", 0, 0,
7269.        { 9087,-2693,-1049,-6715,14382,2537,-2291,2819,7790 } },
7270.    { "Canon PowerShot G3 X", 0, 0,
7271.        { 9701,-3857,-921,-3149,11537,1817,-786,1817,5147 } },
7272.    { "Canon PowerShot G3", 0, 0,
7273.        { 9212,-2781,-1073,-6573,14189,2605,-2300,2844,7664 } },
```

```
7274.    { "Canon PowerShot G5 X", 0, 0,
7275.        { 9602,-3823,-937,-2984,11495,1675,-407,1415,5049 } },
7276.    { "Canon PowerShot G5", 0, 0,
7277.        { 9757,-2872,-933,-5972,13861,2301,-1622,2328,7212 } },
7278.    { "Canon PowerShot G6", 0, 0,
7279.        { 9877,-3775,-871,-7613,14807,3072,-1448,1305,7485 } },
7280.    { "Canon PowerShot G7 X", 0, 0,
7281.        { 9602,-3823,-937,-2984,11495,1675,-407,1415,5049 } },
7282.    { "Canon PowerShot G9 X Mark II", 0, 0,
7283.        { 10056,-4131,-944,-2576,11143,1625,-238,1294,5179 } },
7284.    { "Canon PowerShot G9 X", 0, 0,
7285.        { 9602,-3823,-937,-2984,11495,1675,-407,1415,5049 } },
7286.    { "Canon PowerShot G9", 0, 0,
7287.        { 7368,-2141,-598,-5621,13254,2625,-1418,1696,5743 } },
7288.    { "Canon PowerShot Pro1", 0, 0,
7289.        { 10062,-3522,-999,-7643,15117,2730,-765,817,7323 } },
7290.    { "Canon PowerShot Pro70", 34, 0,
7291.        { -4155,9818,1529,3939,-25,4522,-5521,9870,6610,-2238,10873,1342 } },
7292.    { "Canon PowerShot Pro90", 0, 0,
7293.        { -4963,9896,2235,4642,-987,4294,-5162,10011,5859,-1770,11230,577 } },
7294.    { "Canon PowerShot S30", 0, 0,
7295.        { 10566,-3652,-1129,-6552,14662,2006,-2197,2581,7670 } },
7296.    { "Canon PowerShot S40", 0, 0,
7297.        { 8510,-2487,-940,-6869,14231,2900,-2318,2829,9013 } },
7298.    { "Canon PowerShot S45", 0, 0,
7299.        { 8163,-2333,-955,-6682,14174,2751,-2077,2597,8041 } },
7300.    { "Canon PowerShot S50", 0, 0,
7301.        { 8882,-2571,-863,-6348,14234,2288,-1516,2172,6569 } },
7302.    { "Canon PowerShot S60", 0, 0,
7303.        { 8795,-2482,-797,-7804,15403,2573,-1422,1996,7082 } },
7304.    { "Canon PowerShot S70", 0, 0,
7305.        { 9976,-3810,-832,-7115,14463,2906,-901,989,7889 } },
7306.    { "Canon PowerShot S90", 0, 0,
7307.        { 12374,-5016,-1049,-1677,9902,2078,-83,852,4683 } },
7308.    { "Canon PowerShot S95", 0, 0,
7309.        { 13440,-5896,-1279,-1236,9598,1931,-180,1001,4651 } },
7310.    { "Canon PowerShot S100", 0, 0,
7311.        { 7968,-2565,-636,-2873,10697,2513,180,667,4211 } },
7312.    { "Canon PowerShot S110", 0, 0,
7313.        { 8039,-2643,-654,-3783,11230,2930,-206,690,4194 } },
7314.    { "Canon PowerShot S120", 0, 0,
7315.        { 6961,-1685,-695,-4625,12945,1836,-1114,2152,5518 } },
7316.    { "Canon PowerShot SX1 IS", 0, 0,
7317.        { 6578,-259,-502,-5974,13030,3309,-308,1058,4970 } },
7318.    { "Canon PowerShot SX50 HS", 0, 0,
7319.        { 12432,-4753,-1247,-2110,10691,1629,-412,1623,4926 } },
7320.    { "Canon PowerShot SX60 HS", 0, 0,
7321.        { 13161,-5451,-1344,-1989,10654,1531,-47,1271,4955 } },
7322.    { "Canon PowerShot A3300", 0, 0,    /* DJC */
7323.        { 10826,-3654,-1023,-3215,11310,1906,0,999,4960 } },
7324.    { "Canon PowerShot A470", 0, 0,     /* DJC */
7325.        { 12513,-4407,-1242,-2680,10276,2405,-878,2215,4734 } },
7326.    { "Canon PowerShot A610", 0, 0,     /* DJC */
7327.        { 15591,-6402,-1592,-5365,13198,2168,-1300,1824,5075 } },
7328.    { "Canon PowerShot A620", 0, 0,     /* DJC */
7329.        { 15265,-6193,-1558,-4125,12116,2010,-888,1639,5220 } },
7330.    { "Canon PowerShot A630", 0, 0,     /* DJC */
7331.        { 14201,-5308,-1757,-6087,14472,1617,-2191,3105,5348 } },
7332.    { "Canon PowerShot A640", 0, 0,     /* DJC */
7333.        { 13124,-5329,-1390,-3602,11658,1944,-1612,2863,4885 } },
7334.    { "Canon PowerShot A650", 0, 0,     /* DJC */
7335.        { 9427,-3036,-959,-2581,10671,1911,-1039,1982,4430 } },
7336.    { "Canon PowerShot A720", 0, 0,     /* DJC */
7337.        { 14573,-5482,-1546,-1266,9799,1468,-1040,1912,3810 } },
7338.    { "Canon PowerShot S3 IS", 0, 0,    /* DJC */
```

```
7339.        { 14062,-5199,-1446,-4712,12470,2243,-1286,2028,4836 } },
7340.    { "Canon PowerShot SX110 IS", 0, 0, /* DJC */
7341.        { 14134,-5576,-1527,-1991,10719,1273,-1158,1929,3581 } },
7342.    { "Canon PowerShot SX220", 0, 0,    /* DJC */
7343.        { 13898,-5076,-1447,-1405,10109,1297,-244,1860,3687 } },
7344.    { "Canon IXUS 160", 0, 0,           /* DJC */
7345.        { 11657,-3781,-1136,-3544,11262,2283,-160,1219,4700 } },
7346.    { "Casio EX-S20", 0, 0,             /* DJC */
7347.        { 11634,-3924,-1128,-4968,12954,2015,-1588,2648,7206 } },
7348.    { "Casio EX-Z750", 0, 0,            /* DJC */
7349.        { 10819,-3873,-1099,-4903,13730,1175,-1755,3751,4632 } },
7350.    { "Casio EX-Z10", 128, 0xfff,       /* DJC */
7351.        { 9790,-3338,-603,-2321,10222,2099,-344,1273,4799 } },
7352.    { "CINE 650", 0, 0,
7353.        { 3390,480,-500,-800,3610,340,-550,2336,1192 } },
7354.    { "CINE 660", 0, 0,
7355.        { 3390,480,-500,-800,3610,340,-550,2336,1192 } },
7356.    { "CINE", 0, 0,
7357.        { 20183,-4295,-423,-3940,15330,3985,-280,4870,9800 } },
7358.    { "Contax N Digital", 0, 0xf1e,
7359.        { 7777,1285,-1053,-9280,16543,2916,-3677,5679,7060 } },
7360.    { "DXO ONE", 0, 0,
7361.        { 6596,-2079,-562,-4782,13016,1933,-970,1581,5181 } },
7362.    { "Epson R-D1", 0, 0,
7363.        { 6827,-1878,-732,-8429,16012,2564,-704,592,7145 } },
7364.    { "Fujifilm E550", 0, 0,
7365.        { 11044,-3888,-1120,-7248,15168,2208,-1531,2277,8069 } },
7366.    { "Fujifilm E900", 0, 0,
7367.        { 9183,-2526,-1078,-7461,15071,2574,-2022,2440,8639 } },
7368.    { "Fujifilm F5", 0, 0,
7369.        { 13690,-5358,-1474,-3369,11600,1998,-132,1554,4395 } },
7370.    { "Fujifilm F6", 0, 0,
7371.        { 13690,-5358,-1474,-3369,11600,1998,-132,1554,4395 } },
7372.    { "Fujifilm F77", 0, 0xfe9,
7373.        { 13690,-5358,-1474,-3369,11600,1998,-132,1554,4395 } },
7374.    { "Fujifilm F7", 0, 0,
7375.        { 10004,-3219,-1201,-7036,15047,2107,-1863,2565,7736 } },
7376.    { "Fujifilm F8", 0, 0,
7377.        { 13690,-5358,-1474,-3369,11600,1998,-132,1554,4395 } },
7378.    { "Fujifilm GFX 50S", 0, 0,
7379.        { 11756,-4754,-874,-3056,11045,2305,-381,1457,6006 } },
7380.    { "Fujifilm S100FS", 514, 0,
7381.        { 11521,-4355,-1065,-6524,13767,3058,-1466,1984,6045 } },
7382.    { "Fujifilm S1", 0, 0,
7383.        { 12297,-4882,-1202,-2106,10691,1623,-88,1312,4790 } },
7384.    { "Fujifilm S20Pro", 0, 0,
7385.        { 10004,-3219,-1201,-7036,15047,2107,-1863,2565,7736 } },
7386.    { "Fujifilm S20", 512, 0x3fff,
7387.        { 11401,-4498,-1312,-5088,12751,2613,-838,1568,5941 } },
7388.    { "Fujifilm S2Pro", 128, 0xf15,
7389.        { 12492,-4690,-1402,-7033,15423,1647,-1507,2111,7697 } },
7390.    { "Fujifilm S3Pro", 0, 0x3dff,
7391.        { 11807,-4612,-1294,-8927,16968,1988,-2120,2741,8006 } },
7392.    { "Fujifilm S5Pro", 0, 0,
7393.        { 12300,-5110,-1304,-9117,17143,1998,-1947,2448,8100 } },
7394.    { "Fujifilm S5000", 0, 0,
7395.        { 8754,-2732,-1019,-7204,15069,2276,-1702,2334,6982 } },
7396.    { "Fujifilm S5100", 0, 0,
7397.        { 11940,-4431,-1255,-6766,14428,2542,-993,1165,7421 } },
7398.    { "Fujifilm S5500", 0, 0,
7399.        { 11940,-4431,-1255,-6766,14428,2542,-993,1165,7421 } },
7400.    { "Fujifilm S5200", 0, 0,
7401.        { 9636,-2804,-988,-7442,15040,2589,-1803,2311,8621 } },
7402.    { "Fujifilm S5600", 0, 0,
7403.        { 9636,-2804,-988,-7442,15040,2589,-1803,2311,8621 } },
```

```
7404.     { "Fujifilm S6", 0, 0,
7405.         { 12628,-4887,-1401,-6861,14996,1962,-2198,2782,7091 } },
7406.     { "Fujifilm S7000", 0, 0,
7407.         { 10190,-3506,-1312,-7153,15051,2238,-2003,2399,7505 } },
7408.     { "Fujifilm S9000", 0, 0,
7409.         { 10491,-3423,-1145,-7385,15027,2538,-1809,2275,8692 } },
7410.     { "Fujifilm S9500", 0, 0,
7411.         { 10491,-3423,-1145,-7385,15027,2538,-1809,2275,8692 } },
7412.     { "Fujifilm S9100", 0, 0,
7413.         { 12343,-4515,-1285,-7165,14899,2435,-1895,2496,8800 } },
7414.     { "Fujifilm S9600", 0, 0,
7415.         { 12343,-4515,-1285,-7165,14899,2435,-1895,2496,8800 } },
7416.     { "Fujifilm SL1000", 0, 0,
7417.         { 11705,-4262,-1107,-2282,10791,1709,-555,1713,4945 } },
7418.     { "Fujifilm IS-1", 0, 0,
7419.         { 21461,-10807,-1441,-2332,10599,1999,289,875,7703 } },
7420.     { "Fujifilm IS Pro", 0, 0,
7421.         { 12300,-5110,-1304,-9117,17143,1998,-1947,2448,8100 } },
7422.     { "Fujifilm HS10 HS11", 0, 0xf68,
7423.         { 12440,-3954,-1183,-1123,9674,1708,-83,1614,4086 } },
7424.     { "Fujifilm HS2", 0, 0xfef,
7425.         { 13690,-5358,-1474,-3369,11600,1998,-132,1554,4395 } },
7426.     { "Fujifilm HS3", 0, 0,
7427.         { 13690,-5358,-1474,-3369,11600,1998,-132,1554,4395 } },
7428.     { "Fujifilm HS50EXR", 0, 0,
7429.         { 12085,-4727,-953,-3257,11489,2002,-511,2046,4592 } },
7430.     { "Fujifilm F900EXR", 0, 0,
7431.         { 12085,-4727,-953,-3257,11489,2002,-511,2046,4592 } },
7432.     { "Fujifilm X100F", 0, 0,
7433.         { 11434,-4948,-1210,-3746,12042,1903,-666,1479,5235 } },
7434.     { "Fujifilm X100S", 0, 0,
7435.         { 10592,-4262,-1008,-3514,11355,2465,-870,2025,6386 } },
7436.     { "Fujifilm X100T", 0, 0,
7437.         { 10592,-4262,-1008,-3514,11355,2465,-870,2025,6386 } },
7438.     { "Fujifilm X100", 0, 0,
7439.         { 12161,-4457,-1069,-5034,12874,2400,-795,1724,6904 } },
7440.     { "Fujifilm X10", 0, 0,
7441.         { 13509,-6199,-1254,-4430,12733,1865,-331,1441,5022 } },
7442.     { "Fujifilm X20", 0, 0,
7443.         { 11768,-4971,-1133,-4904,12927,2183,-480,1723,4605 } },
7444.     { "Fujifilm X30", 0, 0,
7445.         { 12328,-5256,-1144,-4469,12927,1675,-87,1291,4351 } },
7446.     { "Fujifilm X70", 0, 0,
7447.         { 10450,-4329,-878,-3217,11105,2421,-752,1758,6519 } },
7448.     { "Fujifilm X-Pro1", 0, 0,
7449.         { 10413,-3996,-993,-3721,11640,2361,-733,1540,6011 } },
7450.     { "Fujifilm X-Pro2", 0, 0,
7451.         { 11434,-4948,-1210,-3746,12042,1903,-666,1479,5235 } },
7452.     { "Fujifilm X-A10", 0, 0,
7453.         { 11540,-4999,-991,-2949,10963,2278,-382,1049,5605 } },
7454.     { "Fujifilm X-A20", 0, 0,
7455.         { 11540,-4999,-991,-2949,10963,2278,-382,1049,5605 } },
7456.     { "Fujifilm X-A1", 0, 0,
7457.         { 11086,-4555,-839,-3512,11310,2517,-815,1341,5940 } },
7458.     { "Fujifilm X-A2", 0, 0,
7459.         { 10763,-4560,-917,-3346,11311,2322,-475,1135,5843 } },
7460.     { "Fujifilm X-A3", 0, 0,
7461.         { 12407,-5222,-1086,-2971,11116,2120,-294,1029,5284 } },
7462.     { "Fujifilm X-A5", 0, 0,
7463.         { 11673,-4760,-1041,-3988,12058,2166,-771,1417,5569 } },
7464.     { "Fujifilm X-E1", 0, 0,
7465.         { 10413,-3996,-993,-3721,11640,2361,-733,1540,6011 } },
7466.     { "Fujifilm X-E2S", 0, 0,
7467.         { 11562,-5118,-961,-3022,11007,2311,-525,1569,6097 } },
7468.     { "Fujifilm X-E2", 0, 0,
```

```
7469.        { 8458,-2451,-855,-4597,12447,2407,-1475,2482,6526 } },
7470.    { "Fujifilm X-E3", 0, 0,
7471.        { 11434,-4948,-1210,-3746,12042,1903,-666,1479,5235 } },
7472.    { "Fujifilm X-H1", 0, 0,
7473.        { 11434,-4948,-1210,-3746,12042,1903,-666,1479,5235 } },
7474.    { "Fujifilm X-M1", 0, 0,
7475.        { 10413,-3996,-993,-3721,11640,2361,-733,1540,6011 } },
7476.    { "Fujifilm X-S1", 0, 0,
7477.        { 13509,-6199,-1254,-4430,12733,1865,-331,1441,5022 } },
7478.    { "Fujifilm X-T1", 0, 0,    /* also X-T10 */
7479.        { 8458,-2451,-855,-4597,12447,2407,-1475,2482,6526 } },
7480.    { "Fujifilm X-T2", 0, 0,    /* also X-T20 */
7481.        { 11434,-4948,-1210,-3746,12042,1903,-666,1479,5235 } },
7482.    { "Fujifilm XF1", 0, 0,
7483.        { 13509,-6199,-1254,-4430,12733,1865,-331,1441,5022 } },
7484.    { "Fujifilm XQ", 0, 0,      /* XQ1 and XQ2 */
7485.        { 9252,-2704,-1064,-5893,14265,1717,-1101,2341,4349 } },
7486.    { "GoPro HERO5 Black", 0, 0,
7487.        { 10344,-4210,-620,-2315,10625,1948,93,1058,5541 } },
7488.    { "Imacon Ixpress", 0, 0,           /* DJC */
7489.        { 7025,-1415,-704,-5188,13765,1424,-1248,2742,6038 } },
7490.    { "Kodak NC2000", 0, 0,
7491.        { 13891,-6055,-803,-465,9919,642,2121,82,1291 } },
7492.    { "Kodak DCS315C", 8, 0,
7493.        { 17523,-4827,-2510,756,8546,-137,6113,1649,2250 } },
7494.    { "Kodak DCS330C", 8, 0,
7495.        { 20620,-7572,-2801,-103,10073,-396,3551,-233,2220 } },
7496.    { "Kodak DCS420", 0, 0,
7497.        { 10868,-1852,-644,-1537,11083,484,2343,628,2216 } },
7498.    { "Kodak DCS460", 0, 0,
7499.        { 10592,-2206,-967,-1944,11685,230,2206,670,1273 } },
7500.    { "Kodak EOSDCS1", 0, 0,
7501.        { 10592,-2206,-967,-1944,11685,230,2206,670,1273 } },
7502.    { "Kodak EOSDCS3B", 0, 0,
7503.        { 9898,-2700,-940,-2478,12219,206,1985,634,1031 } },
7504.    { "Kodak DCS520C", 178, 0,
7505.        { 24542,-10860,-3401,-1490,11370,-297,2858,-605,3225 } },
7506.    { "Kodak DCS560C", 177, 0,
7507.        { 20482,-7172,-3125,-1033,10410,-285,2542,226,3136 } },
7508.    { "Kodak DCS620C", 177, 0,
7509.        { 23617,-10175,-3149,-2054,11749,-272,2586,-489,3453 } },
7510.    { "Kodak DCS620X", 176, 0,
7511.        { 13095,-6231,154,12221,-21,-2137,895,4602,2258 } },
7512.    { "Kodak DCS660C", 173, 0,
7513.        { 18244,-6351,-2739,-791,11193,-521,3711,-129,2802 } },
7514.    { "Kodak DCS720X", 0, 0,
7515.        { 11775,-5884,950,9556,1846,-1286,-1019,6221,2728 } },
7516.    { "Kodak DCS760C", 0, 0,
7517.        { 16623,-6309,-1411,-4344,13923,323,2285,274,2926 } },
7518.    { "Kodak DCS Pro SLR", 0, 0,
7519.        { 5494,2393,-232,-6427,13850,2846,-1876,3997,5445 } },
7520.    { "Kodak DCS Pro 14nx", 0, 0,
7521.        { 5494,2393,-232,-6427,13850,2846,-1876,3997,5445 } },
7522.    { "Kodak DCS Pro 14", 0, 0,
7523.        { 7791,3128,-776,-8588,16458,2039,-2455,4006,6198 } },
7524.    { "Kodak ProBack645", 0, 0,
7525.        { 16414,-6060,-1470,-3555,13037,473,2545,122,4948 } },
7526.    { "Kodak ProBack", 0, 0,
7527.        { 21179,-8316,-2918,-915,11019,-165,3477,-180,4210 } },
7528.    { "Kodak P712", 0, 0,
7529.        { 9658,-3314,-823,-5163,12695,2768,-1342,1843,6044 } },
7530.    { "Kodak P850", 0, 0xf7c,
7531.        { 10511,-3836,-1102,-6946,14587,2558,-1481,1792,6246 } },
7532.    { "Kodak P880", 0, 0xfff,
7533.        { 12805,-4662,-1376,-7480,15267,2360,-1626,2194,7904 } },
```

```
7534.    { "Kodak EasyShare Z980", 0, 0,
7535.        { 11313,-3559,-1101,-3893,11891,2257,-1214,2398,4908 } },
7536.    { "Kodak EasyShare Z981", 0, 0,
7537.        { 12729,-4717,-1188,-1367,9187,2582,274,860,4411 } },
7538.    { "Kodak EasyShare Z990", 0, 0xfed,
7539.        { 11749,-4048,-1309,-1867,10572,1489,-138,1449,4522 } },
7540.    { "Kodak EASYSHARE Z1015", 0, 0xef1,
7541.        { 11265,-4286,-992,-4694,12343,2647,-1090,1523,5447 } },
7542.    { "Leaf CMost", 0, 0,
7543.        { 3952,2189,449,-6701,14585,2275,-4536,7349,6536 } },
7544.    { "Leaf Valeo 6", 0, 0,
7545.        { 3952,2189,449,-6701,14585,2275,-4536,7349,6536 } },
7546.    { "Leaf Aptus 54S", 0, 0,
7547.        { 8236,1746,-1314,-8251,15953,2428,-3673,5786,5771 } },
7548.    { "Leaf Aptus 65", 0, 0,
7549.        { 7914,1414,-1190,-8777,16582,2280,-2811,4605,5562 } },
7550.    { "Leaf Aptus 75", 0, 0,
7551.        { 7914,1414,-1190,-8777,16582,2280,-2811,4605,5562 } },
7552.    { "Leaf", 0, 0,
7553.        { 8236,1746,-1314,-8251,15953,2428,-3673,5786,5771 } },
7554.    { "Mamiya ZD", 0, 0,
7555.        { 7645,2579,-1363,-8689,16717,2015,-3712,5941,5961 } },
7556.    { "Micron 2010", 110, 0,            /* DJC */
7557.        { 16695,-3761,-2151,155,9682,163,3433,951,4904 } },
7558.    { "Minolta DiMAGE 5", 0, 0xf7d,
7559.        { 8983,-2942,-963,-6556,14476,2237,-2426,2887,8014 } },
7560.    { "Minolta DiMAGE 7Hi", 0, 0xf7d,
7561.        { 11368,-3894,-1242,-6521,14358,2339,-2475,3056,7285 } },
7562.    { "Minolta DiMAGE 7", 0, 0xf7d,
7563.        { 9144,-2777,-998,-6676,14556,2281,-2470,3019,7744 } },
7564.    { "Minolta DiMAGE A1", 0, 0xf8b,
7565.        { 9274,-2547,-1167,-8220,16323,1943,-2273,2720,8340 } },
7566.    { "Minolta DiMAGE A200", 0, 0,
7567.        { 8560,-2487,-986,-8112,15535,2771,-1209,1324,7743 } },
7568.    { "Minolta DiMAGE A2", 0, 0xf8f,
7569.        { 9097,-2726,-1053,-8073,15506,2762,-966,981,7763 } },
7570.    { "Minolta DiMAGE Z2", 0, 0,        /* DJC */
7571.        { 11280,-3564,-1370,-4655,12374,2282,-1423,2168,5396 } },
7572.    { "Minolta DYNAX 5", 0, 0xffb,
7573.        { 10284,-3283,-1086,-7957,15762,2316,-829,882,6644 } },
7574.    { "Minolta DYNAX 7", 0, 0xffb,
7575.        { 10239,-3104,-1099,-8037,15727,2451,-927,925,6871 } },
7576.    { "Motorola PIXL", 0, 0,            /* DJC */
7577.        { 8898,-989,-1033,-3292,11619,1674,-661,3178,5216 } },
7578.    { "Nikon D100", 0, 0,
7579.        { 5902,-933,-782,-8983,16719,2354,-1402,1455,6464 } },
7580.    { "Nikon D1H", 0, 0,
7581.        { 7577,-2166,-926,-7454,15592,1934,-2377,2808,8606 } },
7582.    { "Nikon D1X", 0, 0,
7583.        { 7702,-2245,-975,-9114,17242,1875,-2679,3055,8521 } },
7584.    { "Nikon D1", 0, 0, /* multiplied by 2.218750, 1.0, 1.148438 */
7585.        { 16772,-4726,-2141,-7611,15713,1972,-2846,3494,9521 } },
7586.    { "Nikon D200", 0, 0xfbc,
7587.        { 8367,-2248,-763,-8758,16447,2422,-1527,1550,8053 } },
7588.    { "Nikon D2H", 0, 0,
7589.        { 5710,-901,-615,-8594,16617,2024,-2975,4120,6830 } },
7590.    { "Nikon D2X", 0, 0,
7591.        { 10231,-2769,-1255,-8301,15900,2552,-797,680,7148 } },
7592.    { "Nikon D3000", 0, 0,
7593.        { 8736,-2458,-935,-9075,16894,2251,-1354,1242,8263 } },
7594.    { "Nikon D3100", 0, 0,
7595.        { 7911,-2167,-813,-5327,13150,2408,-1288,2483,7968 } },
7596.    { "Nikon D3200", 0, 0xfb9,
7597.        { 7013,-1408,-635,-5268,12902,2640,-1470,2801,7379 } },
7598.    { "Nikon D3300", 0, 0,
```

```
7599.        { 6988,-1384,-714,-5631,13410,2447,-1485,2204,7318 } },
7600.    { "Nikon D3400", 0, 0,
7601.        { 6988,-1384,-714,-5631,13410,2447,-1485,2204,7318 } },
7602.    { "Nikon D300", 0, 0,
7603.        { 9030,-1992,-715,-8465,16302,2255,-2689,3217,8069 } },
7604.    { "Nikon D3X", 0, 0,
7605.        { 7171,-1986,-648,-8085,15555,2718,-2170,2512,7457 } },
7606.    { "Nikon D3S", 0, 0,
7607.        { 8828,-2406,-694,-4874,12603,2541,-660,1509,7587 } },
7608.    { "Nikon D3", 0, 0,
7609.        { 8139,-2171,-663,-8747,16541,2295,-1925,2008,8093 } },
7610.    { "Nikon D40X", 0, 0,
7611.        { 8819,-2543,-911,-9025,16928,2151,-1329,1213,8449 } },
7612.    { "Nikon D40", 0, 0,
7613.        { 6992,-1668,-806,-8138,15748,2543,-874,850,7897 } },
7614.    { "Nikon D4S", 0, 0,
7615.        { 8598,-2848,-857,-5618,13606,2195,-1002,1773,7137 } },
7616.    { "Nikon D4", 0, 0,
7617.        { 8598,-2848,-857,-5618,13606,2195,-1002,1773,7137 } },
7618.    { "Nikon Df", 0, 0,
7619.        { 8598,-2848,-857,-5618,13606,2195,-1002,1773,7137 } },
7620.    { "Nikon D5000", 0, 0xf00,
7621.        { 7309,-1403,-519,-8474,16008,2622,-2433,2826,8064 } },
7622.    { "Nikon D5100", 0, 0x3de6,
7623.        { 8198,-2239,-724,-4871,12389,2798,-1043,2050,7181 } },
7624.    { "Nikon D5200", 0, 0,
7625.        { 8322,-3112,-1047,-6367,14342,2179,-988,1638,6394 } },
7626.    { "Nikon D5300", 0, 0,
7627.        { 6988,-1384,-714,-5631,13410,2447,-1485,2204,7318 } },
7628.    { "Nikon D5500", 0, 0,
7629.        { 8821,-2938,-785,-4178,12142,2287,-824,1651,6860 } },
7630.    { "Nikon D5600", 0, 0,
7631.        { 8821,-2938,-785,-4178,12142,2287,-824,1651,6860 } },
7632.    { "Nikon D500", 0, 0,
7633.        { 8813,-3210,-1036,-4703,12868,2021,-1054,1940,6129 } },
7634.    { "Nikon D50", 0, 0,
7635.        { 7732,-2422,-789,-8238,15884,2498,-859,783,7330 } },
7636.    { "Nikon D5", 0, 0,
7637.        { 9200,-3522,-992,-5755,13803,2117,-753,1486,6338 } },
7638.    { "Nikon D600", 0, 0x3e07,
7639.        { 8178,-2245,-609,-4857,12394,2776,-1207,2086,7298 } },
7640.    { "Nikon D610", 0, 0,
7641.        { 8178,-2245,-609,-4857,12394,2776,-1207,2086,7298 } },
7642.    { "Nikon D60", 0, 0,
7643.        { 8736,-2458,-935,-9075,16894,2251,-1354,1242,8263 } },
7644.    { "Nikon D7000", 0, 0,
7645.        { 8198,-2239,-724,-4871,12389,2798,-1043,2050,7181 } },
7646.    { "Nikon D7100", 0, 0,
7647.        { 8322,-3112,-1047,-6367,14342,2179,-988,1638,6394 } },
7648.    { "Nikon D7200", 0, 0,
7649.        { 8322,-3112,-1047,-6367,14342,2179,-988,1638,6394 } },
7650.    { "Nikon D7500", 0, 0,
7651.        { 8813,-3210,-1036,-4703,12868,2021,-1054,1940,6129 } },
7652.    { "Nikon D750", 0, 0,
7653.        { 9020,-2890,-715,-4535,12436,2348,-934,1919,7086 } },
7654.    { "Nikon D700", 0, 0,
7655.        { 8139,-2171,-663,-8747,16541,2295,-1925,2008,8093 } },
7656.    { "Nikon D70", 0, 0,
7657.        { 7732,-2422,-789,-8238,15884,2498,-859,783,7330 } },
7658.    { "Nikon D850", 0, 0,
7659.        { 10405,-3755,-1270,-5461,13787,1793,-1040,2015,6785 } },
7660.    { "Nikon D810", 0, 0,
7661.        { 9369,-3195,-791,-4488,12430,2301,-893,1796,6872 } },
7662.    { "Nikon D800", 0, 0,
7663.        { 7866,-2108,-555,-4869,12483,2681,-1176,2069,7501 } },
```

```
7664.    { "Nikon D80", 0, 0,
7665.        { 8629,-2410,-883,-9055,16940,2171,-1490,1363,8520 } },
7666.    { "Nikon D90", 0, 0xf00,
7667.        { 7309,-1403,-519,-8474,16008,2622,-2434,2826,8064 } },
7668.    { "Nikon E700", 0, 0x3dd,           /* DJC */
7669.        { -3746,10611,1665,9621,-1734,2114,-2389,7082,3064,3406,6116,-244 } },
7670.    { "Nikon E800", 0, 0x3dd,           /* DJC */
7671.        { -3746,10611,1665,9621,-1734,2114,-2389,7082,3064,3406,6116,-244 } },
7672.    { "Nikon E950", 0, 0x3dd,           /* DJC */
7673.        { -3746,10611,1665,9621,-1734,2114,-2389,7082,3064,3406,6116,-244 } },
7674.    { "Nikon E995", 0, 0,       /* copied from E5000 */
7675.        { -5547,11762,2189,5814,-558,3342,-4924,9840,5949,688,9083,96 } },
7676.    { "Nikon E2100", 0, 0,      /* copied from Z2, new white balance */
7677.        { 13142,-4152,-1596,-4655,12374,2282,-1769,2696,6711} },
7678.    { "Nikon E2500", 0, 0,
7679.        { -5547,11762,2189,5814,-558,3342,-4924,9840,5949,688,9083,96 } },
7680.    { "Nikon E3200", 0, 0,           /* DJC */
7681.        { 9846,-2085,-1019,-3278,11109,2170,-774,2134,5745 } },
7682.    { "Nikon E4300", 0, 0,      /* copied from Minolta DiMAGE Z2 */
7683.        { 11280,-3564,-1370,-4655,12374,2282,-1423,2168,5396 } },
7684.    { "Nikon E4500", 0, 0,
7685.        { -5547,11762,2189,5814,-558,3342,-4924,9840,5949,688,9083,96 } },
7686.    { "Nikon E5000", 0, 0,
7687.        { -5547,11762,2189,5814,-558,3342,-4924,9840,5949,688,9083,96 } },
7688.    { "Nikon E5400", 0, 0,
7689.        { 9349,-2987,-1001,-7919,15766,2266,-2098,2680,6839 } },
7690.    { "Nikon E5700", 0, 0,
7691.        { -5368,11478,2368,5537,-113,3148,-4969,10021,5782,778,9028,211 } },
7692.    { "Nikon E8400", 0, 0,
7693.        { 7842,-2320,-992,-8154,15718,2599,-1098,1342,7560 } },
7694.    { "Nikon E8700", 0, 0,
7695.        { 8489,-2583,-1036,-8051,15583,2643,-1307,1407,7354 } },
7696.    { "Nikon E8800", 0, 0,
7697.        { 7971,-2314,-913,-8451,15762,2894,-1442,1520,7610 } },
7698.    { "Nikon COOLPIX A", 0, 0,
7699.        { 8198,-2239,-724,-4871,12389,2798,-1043,2050,7181 } },
7700.    { "Nikon COOLPIX B700", 200, 0,
7701.        { 14387,-6014,-1299,-1357,9975,1616,467,1047,4744 } },
7702.    { "Nikon COOLPIX P330", 200, 0,
7703.        { 10321,-3920,-931,-2750,11146,1824,-442,1545,5539 } },
7704.    { "Nikon COOLPIX P340", 200, 0,
7705.        { 10321,-3920,-931,-2750,11146,1824,-442,1545,5539 } },
7706.    { "Nikon COOLPIX P6000", 0, 0,
7707.        { 9698,-3367,-914,-4706,12584,2368,-837,968,5801 } },
7708.    { "Nikon COOLPIX P7000", 0, 0,
7709.        { 11432,-3679,-1111,-3169,11239,2202,-791,1380,4455 } },
7710.    { "Nikon COOLPIX P7100", 0, 0,
7711.        { 11053,-4269,-1024,-1976,10182,2088,-526,1263,4469 } },
7712.    { "Nikon COOLPIX P7700", 200, 0,
7713.        { 10321,-3920,-931,-2750,11146,1824,-442,1545,5539 } },
7714.    { "Nikon COOLPIX P7800", 200, 0,
7715.        { 10321,-3920,-931,-2750,11146,1824,-442,1545,5539 } },
7716.    { "Nikon 1 V3", 0, 0,
7717.        { 5958,-1559,-571,-4021,11453,2939,-634,1548,5087 } },
7718.    { "Nikon 1 J4", 0, 0,
7719.        { 5958,-1559,-571,-4021,11453,2939,-634,1548,5087 } },
7720.    { "Nikon 1 J5", 0, 0,
7721.        { 7520,-2518,-645,-3844,12102,1945,-913,2249,6835 } },
7722.    { "Nikon 1 S2", 200, 0,
7723.        { 6612,-1342,-618,-3338,11055,2623,-174,1792,5075 } },
7724.    { "Nikon 1 V2", 0, 0,
7725.        { 6588,-1305,-693,-3277,10987,2634,-355,2016,5106 } },
7726.    { "Nikon 1 J3", 0, 0,
7727.        { 6588,-1305,-693,-3277,10987,2634,-355,2016,5106 } },
7728.    { "Nikon 1 AW1", 0, 0,
```

```
7729.        { 6588,-1305,-693,-3277,10987,2634,-355,2016,5106 } },
7730.    { "Nikon 1 ", 0, 0,          /* J1, J2, S1, V1 */
7731.        { 8994,-2667,-865,-4594,12324,2552,-699,1786,6260 } },
7732.    { "Olympus AIR A01", 0, 0,
7733.        { 8992,-3093,-639,-2563,10721,2122,-437,1270,5473 } },
7734.    { "Olympus C5050", 0, 0,
7735.        { 10508,-3124,-1273,-6079,14294,1901,-1653,2306,6237 } },
7736.    { "Olympus C5060", 0, 0,
7737.        { 10445,-3362,-1307,-7662,15690,2058,-1135,1176,7602 } },
7738.    { "Olympus C7070", 0, 0,
7739.        { 10252,-3531,-1095,-7114,14850,2436,-1451,1723,6365 } },
7740.    { "Olympus C70", 0, 0,
7741.        { 10793,-3791,-1146,-7498,15177,2488,-1390,1577,7321 } },
7742.    { "Olympus C80", 0, 0,
7743.        { 8606,-2509,-1014,-8238,15714,2703,-942,979,7760 } },
7744.    { "Olympus E-10", 0, 0xffc,
7745.        { 12745,-4500,-1416,-6062,14542,1580,-1934,2256,6603 } },
7746.    { "Olympus E-1", 0, 0,
7747.        { 11846,-4767,-945,-7027,15878,1089,-2699,4122,8311 } },
7748.    { "Olympus E-20", 0, 0xffc,
7749.        { 13173,-4732,-1499,-5807,14036,1895,-2045,2452,7142 } },
7750.    { "Olympus E-300", 0, 0,
7751.        { 7828,-1761,-348,-5788,14071,1830,-2853,4518,6557 } },
7752.    { "Olympus E-330", 0, 0,
7753.        { 8961,-2473,-1084,-7979,15990,2067,-2319,3035,8249 } },
7754.    { "Olympus E-30", 0, 0xfbc,
7755.        { 8144,-1861,-1111,-7763,15894,1929,-1865,2542,7607 } },
7756.    { "Olympus E-3", 0, 0xf99,
7757.        { 9487,-2875,-1115,-7533,15606,2010,-1618,2100,7389 } },
7758.    { "Olympus E-400", 0, 0,
7759.        { 6169,-1483,-21,-7107,14761,2536,-2904,3580,8568 } },
7760.    { "Olympus E-410", 0, 0xf6a,
7761.        { 8856,-2582,-1026,-7761,15766,2082,-2009,2575,7469 } },
7762.    { "Olympus E-420", 0, 0xfd7,
7763.        { 8746,-2425,-1095,-7594,15612,2073,-1780,2309,7416 } },
7764.    { "Olympus E-450", 0, 0xfd2,
7765.        { 8745,-2425,-1095,-7594,15613,2073,-1780,2309,7416 } },
7766.    { "Olympus E-500", 0, 0,
7767.        { 8136,-1968,-299,-5481,13742,1871,-2556,4205,6630 } },
7768.    { "Olympus E-510", 0, 0xf6a,
7769.        { 8785,-2529,-1033,-7639,15624,2112,-1783,2300,7817 } },
7770.    { "Olympus E-520", 0, 0xfd2,
7771.        { 8344,-2322,-1020,-7596,15635,2048,-1748,2269,7287 } },
7772.    { "Olympus E-5", 0, 0xeec,
7773.        { 11200,-3783,-1325,-4576,12593,2206,-695,1742,7504 } },
7774.    { "Olympus E-600", 0, 0xfaf,
7775.        { 8453,-2198,-1092,-7609,15681,2008,-1725,2337,7824 } },
7776.    { "Olympus E-620", 0, 0xfaf,
7777.        { 8453,-2198,-1092,-7609,15681,2008,-1725,2337,7824 } },
7778.    { "Olympus E-P1", 0, 0xffd,
7779.        { 8343,-2050,-1021,-7715,15705,2103,-1831,2380,8235 } },
7780.    { "Olympus E-P2", 0, 0xffd,
7781.        { 8343,-2050,-1021,-7715,15705,2103,-1831,2380,8235 } },
7782.    { "Olympus E-P3", 0, 0,
7783.        { 7575,-2159,-571,-3722,11341,2725,-1434,2819,6271 } },
7784.    { "Olympus E-P5", 0, 0,
7785.        { 8380,-2630,-639,-2887,10725,2496,-627,1427,5438 } },
7786.    { "Olympus E-PL1s", 0, 0,
7787.        { 11409,-3872,-1393,-4572,12757,2003,-709,1810,7415 } },
7788.    { "Olympus E-PL1", 0, 0,
7789.        { 11408,-4289,-1215,-4286,12385,2118,-387,1467,7787 } },
7790.    { "Olympus E-PL2", 0, 0xcf3,
7791.        { 15030,-5552,-1806,-3987,12387,1767,-592,1670,7023 } },
7792.    { "Olympus E-PL3", 0, 0,
7793.        { 7575,-2159,-571,-3722,11341,2725,-1434,2819,6271 } },
```

```
7794.    { "Olympus E-PL5", 0, 0xfcb,
7795.        { 8380,-2630,-639,-2887,10725,2496,-627,1427,5438 } },
7796.    { "Olympus E-PL6", 0, 0,
7797.        { 8380,-2630,-639,-2887,10725,2496,-627,1427,5438 } },
7798.    { "Olympus E-PL7", 0, 0,
7799.        { 9197,-3190,-659,-2606,10830,2039,-458,1250,5458 } },
7800.    { "Olympus E-PL8", 0, 0,
7801.        { 9197,-3190,-659,-2606,10830,2039,-458,1250,5458 } },
7802.    { "Olympus E-PL9", 0, 0,
7803.        { 8380,-2630,-639,-2887,10725,2496,-627,1427,5438 } },
7804.    { "Olympus E-PM1", 0, 0,
7805.        { 7575,-2159,-571,-3722,11341,2725,-1434,2819,6271 } },
7806.    { "Olympus E-PM2", 0, 0,
7807.        { 8380,-2630,-639,-2887,10725,2496,-627,1427,5438 } },
7808.    { "Olympus E-M10", 0, 0,    /* also E-M10 Mark II & III */
7809.        { 8380,-2630,-639,-2887,10725,2496,-627,1427,5438 } },
7810.    { "Olympus E-M1Mark II", 0, 0,
7811.        { 9383,-3170,-763,-2457,10702,2020,-384,1236,5552 } },
7812.    { "Olympus E-M1", 0, 0,
7813.        { 7687,-1984,-606,-4327,11928,2721,-1381,2339,6452 } },
7814.    { "Olympus E-M5MarkII", 0, 0,
7815.        { 9422,-3258,-711,-2655,10898,2015,-512,1354,5512 } },
7816.    { "Olympus E-M5", 0, 0xfe1,
7817.        { 8380,-2630,-639,-2887,10725,2496,-627,1427,5438 } },
7818.    { "Olympus PEN-F", 0, 0,
7819.        { 9476,-3182,-765,-2613,10958,1893,-449,1315,5268 } },
7820.    { "Olympus SH-2", 0, 0,
7821.        { 10156,-3425,-1077,-2611,11177,1624,-385,1592,5080 } },
7822.    { "Olympus SP350", 0, 0,
7823.        { 12078,-4836,-1069,-6671,14306,2578,-786,939,7418 } },
7824.    { "Olympus SP3", 0, 0,
7825.        { 11766,-4445,-1067,-6901,14421,2707,-1029,1217,7572 } },
7826.    { "Olympus SP500UZ", 0, 0xfff,
7827.        { 9493,-3415,-666,-5211,12334,3260,-1548,2262,6482 } },
7828.    { "Olympus SP510UZ", 0, 0xffe,
7829.        { 10593,-3607,-1010,-5881,13127,3084,-1200,1805,6721 } },
7830.    { "Olympus SP550UZ", 0, 0xffe,
7831.        { 11597,-4006,-1049,-5432,12799,2957,-1029,1750,6516 } },
7832.    { "Olympus SP560UZ", 0, 0xff9,
7833.        { 10915,-3677,-982,-5587,12986,2911,-1168,1968,6223 } },
7834.    { "Olympus SP570UZ", 0, 0,
7835.        { 11522,-4044,-1146,-4736,12172,2904,-988,1829,6039 } },
7836.    { "Olympus STYLUS1", 0, 0,
7837.        { 8360,-2420,-880,-3928,12353,1739,-1381,2416,5173 } },
7838.    { "Olympus TG-4", 0, 0,
7839.        { 11426,-4159,-1126,-2066,10678,1593,-120,1327,4998 } },
7840.    { "Olympus TG-5", 0, 0,
7841.        { 10899,-3833,-1082,-2112,10736,1575,-267,1452,5269 } },
7842.    { "Olympus XZ-10", 0, 0,
7843.        { 9777,-3483,-925,-2886,11297,1800,-602,1663,5134 } },
7844.    { "Olympus XZ-1", 0, 0,
7845.        { 10901,-4095,-1074,-1141,9208,2293,-62,1417,5158 } },
7846.    { "Olympus XZ-2", 0, 0,
7847.        { 9777,-3483,-925,-2886,11297,1800,-602,1663,5134 } },
7848.    { "OmniVision", 0, 0,               /* DJC */
7849.        { 12782,-4059,-379,-478,9066,1413,1340,1513,5176 } },
7850.    { "Pentax *ist DL2", 0, 0,
7851.        { 10504,-2438,-1189,-8603,16207,2531,-1022,863,12242 } },
7852.    { "Pentax *ist DL", 0, 0,
7853.        { 10829,-2838,-1115,-8339,15817,2696,-837,680,11939 } },
7854.    { "Pentax *ist DS2", 0, 0,
7855.        { 10504,-2438,-1189,-8603,16207,2531,-1022,863,12242 } },
7856.    { "Pentax *ist DS", 0, 0,
7857.        { 10371,-2333,-1206,-8688,16231,2602,-1230,1116,11282 } },
7858.    { "Pentax *ist D", 0, 0,
```

```
7859.        { 9651,-2059,-1189,-8881,16512,2487,-1460,1345,10687 } },
7860.    { "Pentax K10D", 0, 0,
7861.        { 9566,-2863,-803,-7170,15172,2112,-818,803,9705 } },
7862.    { "Pentax K1", 0, 0,
7863.        { 11095,-3157,-1324,-8377,15834,2720,-1108,947,11688 } },
7864.    { "Pentax K20D", 0, 0,
7865.        { 9427,-2714,-868,-7493,16092,1373,-2199,3264,7180 } },
7866.    { "Pentax K200D", 0, 0,
7867.        { 9186,-2678,-907,-8693,16517,2260,-1129,1094,8524 } },
7868.    { "Pentax K2000", 0, 0,
7869.        { 11057,-3604,-1155,-5152,13046,2329,-282,375,8104 } },
7870.    { "Pentax K-m", 0, 0,
7871.        { 11057,-3604,-1155,-5152,13046,2329,-282,375,8104 } },
7872.    { "Pentax K-x", 0, 0,
7873.        { 8843,-2837,-625,-5025,12644,2668,-411,1234,7410 } },
7874.    { "Pentax K-r", 0, 0,
7875.        { 9895,-3077,-850,-5304,13035,2521,-883,1768,6936 } },
7876.    { "Pentax K-1", 0, 0,
7877.        { 8596,-2981,-639,-4202,12046,2431,-685,1424,6122 } },
7878.    { "Pentax K-30", 0, 0,
7879.        { 8710,-2632,-1167,-3995,12301,1881,-981,1719,6535 } },
7880.    { "Pentax K-3 II", 0, 0,
7881.        { 8626,-2607,-1155,-3995,12301,1881,-1039,1822,6925 } },
7882.    { "Pentax K-3", 0, 0,
7883.        { 7415,-2052,-721,-5186,12788,2682,-1446,2157,6773 } },
7884.    { "Pentax K-5 II", 0, 0,
7885.        { 8170,-2725,-639,-4440,12017,2744,-771,1465,6599 } },
7886.    { "Pentax K-5", 0, 0,
7887.        { 8713,-2833,-743,-4342,11900,2772,-722,1543,6247 } },
7888.    { "Pentax K-70", 0, 0,
7889.        { 8270,-2117,-1299,-4359,12953,1515,-1078,1933,5975 } },
7890.    { "Pentax K-7", 0, 0,
7891.        { 9142,-2947,-678,-8648,16967,1663,-2224,2898,8615 } },
7892.    { "Pentax K-S1", 0, 0,
7893.        { 8512,-3211,-787,-4167,11966,2487,-638,1288,6054 } },
7894.    { "Pentax K-S2", 0, 0,
7895.        { 8662,-3280,-798,-3928,11771,2444,-586,1232,6054 } },
7896.    { "Pentax KP", 0, 0,
7897.        { 8617,-3228,-1034,-4674,12821,2044,-803,1577,5728 } },
7898.    { "Pentax Q-S1", 0, 0,
7899.        { 12995,-5593,-1107,-1879,10139,2027,-64,1233,4919 } },
7900.    { "Pentax 645D", 0, 0x3e00,
7901.        { 10646,-3593,-1158,-3329,11699,1831,-667,2874,6287 } },
7902.    { "Panasonic DMC-CM1", 15, 0,
7903.        { 8770,-3194,-820,-2871,11281,1803,-513,1552,4434 } },
7904.    { "Panasonic DC-FZ80", 0, 0,
7905.        { 8550,-2908,-842,-3195,11529,1881,-338,1603,4631 } },
7906.    { "Panasonic DMC-FZ8", 0, 0xf7f,
7907.        { 8986,-2755,-802,-6341,13575,3077,-1476,2144,6379 } },
7908.    { "Panasonic DMC-FZ18", 0, 0,
7909.        { 9932,-3060,-935,-5809,13331,2753,-1267,2155,5575 } },
7910.    { "Panasonic DMC-FZ28", 15, 0xf96,
7911.        { 10109,-3488,-993,-5412,12812,2916,-1305,2140,5543 } },
7912.    { "Panasonic DMC-FZ2500", 15, 0,
7913.        { 7386,-2443,-743,-3437,11864,1757,-608,1660,4766 } },
7914.    { "Panasonic DMC-FZ330", 15, 0,
7915.        { 8378,-2798,-769,-3068,11410,1877,-538,1792,4623 } },
7916.    { "Panasonic DMC-FZ300", 15, 0,
7917.        { 8378,-2798,-769,-3068,11410,1877,-538,1792,4623 } },
7918.    { "Panasonic DMC-FZ30", 0, 0xf94,
7919.        { 10976,-4029,-1141,-7918,15491,2600,-1670,2071,8246 } },
7920.    { "Panasonic DMC-FZ3", 15, 0,       /* FZ35, FZ38 */
7921.        { 9938,-2780,-890,-4604,12393,2480,-1117,2304,4620 } },
7922.    { "Panasonic DMC-FZ4", 15, 0,       /* FZ40, FZ45 */
7923.        { 13639,-5535,-1371,-1698,9633,2430,316,1152,4108 } },
```

```
7924.    { "Panasonic DMC-FZ50", 0, 0,
7925.        { 7906,-2709,-594,-6231,13351,3220,-1922,2631,6537 } },
7926.    { "Panasonic DMC-FZ7", 15, 0,       /* FZ70, FZ72 */
7927.        { 11532,-4324,-1066,-2375,10847,1749,-564,1699,4351 } },
7928.    { "Leica V-LUX1", 0, 0,
7929.        { 7906,-2709,-594,-6231,13351,3220,-1922,2631,6537 } },
7930.    { "Panasonic DMC-L10", 15, 0xf96,
7931.        { 8025,-1942,-1050,-7920,15904,2100,-2456,3005,7039 } },
7932.    { "Panasonic DMC-L1", 0, 0xf7f,
7933.        { 8054,-1885,-1025,-8349,16367,2040,-2805,3542,7629 } },
7934.    { "Leica DIGILUX 3", 0, 0xf7f,
7935.        { 8054,-1885,-1025,-8349,16367,2040,-2805,3542,7629 } },
7936.    { "Panasonic DMC-LC1", 0, 0,
7937.        { 11340,-4069,-1275,-7555,15266,2448,-2960,3426,7685 } },
7938.    { "Leica DIGILUX 2", 0, 0,
7939.        { 11340,-4069,-1275,-7555,15266,2448,-2960,3426,7685 } },
7940.    { "Panasonic DMC-LX100", 15, 0,
7941.        { 8844,-3538,-768,-3709,11762,2200,-698,1792,5220 } },
7942.    { "Leica D-LUX (Typ 109)", 15, 0,
7943.        { 8844,-3538,-768,-3709,11762,2200,-698,1792,5220 } },
7944.    { "Panasonic DMC-LF1", 15, 0,
7945.        { 9379,-3267,-816,-3227,11560,1881,-926,1928,5340 } },
7946.    { "Leica C (Typ 112)", 15, 0,
7947.        { 9379,-3267,-816,-3227,11560,1881,-926,1928,5340 } },
7948.    { "Panasonic DMC-LX1", 0, 0xf7f,
7949.        { 10704,-4187,-1230,-8314,15952,2501,-920,945,8927 } },
7950.    { "Leica D-LUX2", 0, 0xf7f,
7951.        { 10704,-4187,-1230,-8314,15952,2501,-920,945,8927 } },
7952.    { "Panasonic DMC-LX2", 0, 0,
7953.        { 8048,-2810,-623,-6450,13519,3272,-1700,2146,7049 } },
7954.    { "Leica D-LUX3", 0, 0,
7955.        { 8048,-2810,-623,-6450,13519,3272,-1700,2146,7049 } },
7956.    { "Panasonic DMC-LX3", 15, 0,
7957.        { 8128,-2668,-655,-6134,13307,3161,-1782,2568,6083 } },
7958.    { "Leica D-LUX 4", 15, 0,
7959.        { 8128,-2668,-655,-6134,13307,3161,-1782,2568,6083 } },
7960.    { "Panasonic DMC-LX5", 15, 0,
7961.        { 10909,-4295,-948,-1333,9306,2399,22,1738,4582 } },
7962.    { "Leica D-LUX 5", 15, 0,
7963.        { 10909,-4295,-948,-1333,9306,2399,22,1738,4582 } },
7964.    { "Panasonic DMC-LX7", 15, 0,
7965.        { 10148,-3743,-991,-2837,11366,1659,-701,1893,4899 } },
7966.    { "Leica D-LUX 6", 15, 0,
7967.        { 10148,-3743,-991,-2837,11366,1659,-701,1893,4899 } },
7968.    { "Panasonic DMC-LX9", 15, 0,
7969.        { 7790,-2736,-755,-3452,11870,1769,-628,1647,4898 } },
7970.    { "Panasonic DMC-FZ1000", 15, 0,
7971.        { 7830,-2696,-763,-3325,11667,1866,-641,1712,4824 } },
7972.    { "Leica V-LUX (Typ 114)", 15, 0,
7973.        { 7830,-2696,-763,-3325,11667,1866,-641,1712,4824 } },
7974.    { "Panasonic DMC-FZ100", 15, 0xfff,
7975.        { 16197,-6146,-1761,-2393,10765,1869,366,2238,5248 } },
7976.    { "Leica V-LUX 2", 15, 0xfff,
7977.        { 16197,-6146,-1761,-2393,10765,1869,366,2238,5248 } },
7978.    { "Panasonic DMC-FZ150", 15, 0xfff,
7979.        { 11904,-4541,-1189,-2355,10899,1662,-296,1586,4289 } },
7980.    { "Leica V-LUX 3", 15, 0xfff,
7981.        { 11904,-4541,-1189,-2355,10899,1662,-296,1586,4289 } },
7982.    { "Panasonic DMC-FZ200", 15, 0xfff,
7983.        { 8112,-2563,-740,-3730,11784,2197,-941,2075,4933 } },
7984.    { "Leica V-LUX 4", 15, 0xfff,
7985.        { 8112,-2563,-740,-3730,11784,2197,-941,2075,4933 } },
7986.    { "Panasonic DMC-FX150", 15, 0xfff,
7987.        { 9082,-2907,-925,-6119,13377,3058,-1797,2641,5609 } },
7988.    { "Panasonic DMC-G10", 0, 0,
```

```
7989.        { 10113,-3400,-1114,-4765,12683,2317,-377,1437,6710 } },
7990.    { "Panasonic DMC-G1", 15, 0xf94,
7991.        { 8199,-2065,-1056,-8124,16156,2033,-2458,3022,7220 } },
7992.    { "Panasonic DMC-G2", 15, 0xf3c,
7993.        { 10113,-3400,-1114,-4765,12683,2317,-377,1437,6710 } },
7994.    { "Panasonic DMC-G3", 15, 0xfff,
7995.        { 6763,-1919,-863,-3868,11515,2684,-1216,2387,5879 } },
7996.    { "Panasonic DMC-G5", 15, 0xfff,
7997.        { 7798,-2562,-740,-3879,11584,2613,-1055,2248,5434 } },
7998.    { "Panasonic DMC-G6", 15, 0xfff,
7999.        { 8294,-2891,-651,-3869,11590,2595,-1183,2267,5352 } },
8000.    { "Panasonic DMC-G7", 15, 0xfff,
8001.        { 7610,-2780,-576,-4614,12195,2733,-1375,2393,6490 } },
8002.    { "Panasonic DMC-G8", 15, 0xfff,    /* G8, G80, G81, G85 */
8003.        { 7610,-2780,-576,-4614,12195,2733,-1375,2393,6490 } },
8004.    { "Panasonic DC-G9", 15, 0xfff,
8005.        { 7685,-2375,-634,-3687,11700,2249,-748,1546,5111 } },
8006.    { "Panasonic DMC-GF1", 15, 0xf92,
8007.        { 7888,-1902,-1011,-8106,16085,2099,-2353,2866,7330 } },
8008.    { "Panasonic DMC-GF2", 15, 0xfff,
8009.        { 7888,-1902,-1011,-8106,16085,2099,-2353,2866,7330 } },
8010.    { "Panasonic DMC-GF3", 15, 0xfff,
8011.        { 9051,-2468,-1204,-5212,13276,2121,-1197,2510,6890 } },
8012.    { "Panasonic DMC-GF5", 15, 0xfff,
8013.        { 8228,-2945,-660,-3938,11792,2430,-1094,2278,5793 } },
8014.    { "Panasonic DMC-GF6", 15, 0,
8015.        { 8130,-2801,-946,-3520,11289,2552,-1314,2511,5791 } },
8016.    { "Panasonic DMC-GF7", 15, 0,
8017.        { 7610,-2780,-576,-4614,12195,2733,-1375,2393,6490 } },
8018.    { "Panasonic DMC-GF8", 15, 0,
8019.        { 7610,-2780,-576,-4614,12195,2733,-1375,2393,6490 } },
8020.    { "Panasonic DC-GF9", 15, 0,
8021.        { 7610,-2780,-576,-4614,12195,2733,-1375,2393,6490 } },
8022.    { "Panasonic DMC-GH1", 15, 0xf92,
8023.        { 6299,-1466,-532,-6535,13852,2969,-2331,3112,5984 } },
8024.    { "Panasonic DMC-GH2", 15, 0xf95,
8025.        { 7780,-2410,-806,-3913,11724,2484,-1018,2390,5298 } },
8026.    { "Panasonic DMC-GH3", 15, 0,
8027.        { 6559,-1752,-491,-3672,11407,2586,-962,1875,5130 } },
8028.    { "Panasonic DMC-GH4", 15, 0,
8029.        { 7122,-2108,-512,-3155,11201,2231,-541,1423,5045 } },
8030.    { "Panasonic DC-GH5S", 15, 0,
8031.        { 6929,-2355,-708,-4192,12534,1828,-1097,1989,5195 } },
8032.    { "Panasonic DC-GH5", 15, 0,
8033.        { 7641,-2336,-605,-3218,11299,2187,-485,1338,5121 } },
8034.    { "Panasonic DMC-GM1", 15, 0,
8035.        { 6770,-1895,-744,-5232,13145,2303,-1664,2691,5703 } },
8036.    { "Panasonic DMC-GM5", 15, 0,
8037.        { 8238,-3244,-679,-3921,11814,2384,-836,2022,5852 } },
8038.    { "Panasonic DMC-GX1", 15, 0,
8039.        { 6763,-1919,-863,-3868,11515,2684,-1216,2387,5879 } },
8040.    { "Panasonic DMC-GX7", 15, 0,
8041.        { 7610,-2780,-576,-4614,12195,2733,-1375,2393,6490 } },
8042.    { "Panasonic DMC-GX85", 15, 0,
8043.        { 7771,-3020,-629,-4029,11950,2345,-821,1977,6119 } },
8044.    { "Panasonic DMC-GX8", 15, 0,
8045.        { 7564,-2263,-606,-3148,11239,2177,-540,1435,4853 } },
8046.    { "Panasonic DC-GX9", 15, 0,
8047.        { 7564,-2263,-606,-3148,11239,2177,-540,1435,4853 } },
8048.    { "Panasonic DMC-ZS100", 15, 0,
8049.        { 7790,-2736,-755,-3452,11870,1769,-628,1647,4898 } },
8050.    { "Panasonic DC-ZS200", 15, 0,
8051.        { 7790,-2736,-755,-3452,11870,1769,-628,1647,4898 } },
8052.    { "Panasonic DMC-ZS40", 15, 0,
8053.        { 8607,-2822,-808,-3755,11930,2049,-820,2060,5224 } },
```

```
8054.    { "Panasonic DMC-ZS50", 15, 0,
8055.        { 8802,-3135,-789,-3151,11468,1904,-550,1745,4810 } },
8056.    { "Panasonic DMC-TZ82", 15, 0,
8057.        { 8550,-2908,-842,-3195,11529,1881,-338,1603,4631 } },
8058.    { "Panasonic DMC-ZS6", 15, 0,
8059.        { 8550,-2908,-842,-3195,11529,1881,-338,1603,4631 } },
8060.    { "Panasonic DMC-ZS70", 15, 0,
8061.        { 9052,-3117,-883,-3045,11346,1927,-205,1520,4730 } },
8062.    { "Leica S (Typ 007)", 0, 0,
8063.        { 6063,-2234,-231,-5210,13787,1500,-1043,2866,6997 } },
8064.    { "Leica X", 0, 0,          /* X and X-U, both (Typ 113) */
8065.        { 7712,-2059,-653,-3882,11494,2726,-710,1332,5958 } },
8066.    { "Leica Q (Typ 116)", 0, 0,
8067.        { 11865,-4523,-1441,-5423,14458,935,-1587,2687,4830 } },
8068.    { "Leica M (Typ 262)", 0, 0,
8069.        { 6653,-1486,-611,-4221,13303,929,-881,2416,7226 } },
8070.    { "Leica SL (Typ 601)", 0, 0,
8071.        { 11865,-4523,-1441,-5423,14458,935,-1587,2687,4830 } },
8072.    { "Leica TL2", 0, 0,
8073.        { 5836,-1626,-647,-5384,13326,2261,-1207,2129,5861 } },
8074.    { "Leica TL", 0, 0,
8075.        { 5463,-988,-364,-4634,12036,2946,-766,1389,6522 } },
8076.    { "Leica CL", 0, 0,
8077.        { 7414,-2393,-840,-5127,13180,2138,-1585,2468,5064 } },
8078.    { "Leica M10", 0, 0,
8079.        { 8249,-2849,-620,-5415,14756,565,-957,3074,6517 } },
8080.    { "Phase One H 20", 0, 0,           /* DJC */
8081.        { 1313,1855,-109,-6715,15908,808,-327,1840,6020 } },
8082.    { "Phase One H 25", 0, 0,
8083.        { 2905,732,-237,-8134,16626,1476,-3038,4253,7517 } },
8084.    { "Phase One P 2", 0, 0,
8085.        { 2905,732,-237,-8134,16626,1476,-3038,4253,7517 } },
8086.    { "Phase One P 30", 0, 0,
8087.        { 4516,-245,-37,-7020,14976,2173,-3206,4671,7087 } },
8088.    { "Phase One P 45", 0, 0,
8089.        { 5053,-24,-117,-5684,14076,1702,-2619,4492,5849 } },
8090.    { "Phase One P40", 0, 0,
8091.        { 8035,435,-962,-6001,13872,2320,-1159,3065,5434 } },
8092.    { "Phase One P65", 0, 0,
8093.        { 8035,435,-962,-6001,13872,2320,-1159,3065,5434 } },
8094.    { "Photron BC2-HD", 0, 0,           /* DJC */
8095.        { 14603,-4122,-528,-1810,9794,2017,-297,2763,5936 } },
8096.    { "Red One", 704, 0xffff,           /* DJC */
8097.        { 21014,-7891,-2613,-3056,12201,856,-2203,5125,8042 } },
8098.    { "Ricoh GR II", 0, 0,
8099.        { 4630,-834,-423,-4977,12805,2417,-638,1467,6115 } },
8100.    { "Ricoh GR", 0, 0,
8101.        { 3708,-543,-160,-5381,12254,3556,-1471,1929,8234 } },
8102.    { "Samsung EX1", 0, 0x3e00,
8103.        { 8898,-2498,-994,-3144,11328,2066,-760,1381,4576 } },
8104.    { "Samsung EX2F", 0, 0x7ff,
8105.        { 10648,-3897,-1055,-2022,10573,1668,-492,1611,4742 } },
8106.    { "Samsung EK-GN120", 0, 0,
8107.        { 7557,-2522,-739,-4679,12949,1894,-840,1777,5311 } },
8108.    { "Samsung NX mini", 0, 0,
8109.        { 5222,-1196,-550,-6540,14649,2009,-1666,2819,5657 } },
8110.    { "Samsung NX3300", 0, 0,
8111.        { 8060,-2933,-761,-4504,12890,1762,-630,1489,5227 } },
8112.    { "Samsung NX3000", 0, 0,
8113.        { 8060,-2933,-761,-4504,12890,1762,-630,1489,5227 } },
8114.    { "Samsung NX30", 0, 0,     /* NX30, NX300, NX300M */
8115.        { 7557,-2522,-739,-4679,12949,1894,-840,1777,5311 } },
8116.    { "Samsung NX2000", 0, 0,
8117.        { 7557,-2522,-739,-4679,12949,1894,-840,1777,5311 } },
8118.    { "Samsung NX2", 0, 0xfff,  /* NX20, NX200, NX210 */
```

```
8119.        { 6933,-2268,-753,-4921,13387,1647,-803,1641,6096 } },
8120.    { "Samsung NX1000", 0, 0,
8121.        { 6933,-2268,-753,-4921,13387,1647,-803,1641,6096 } },
8122.    { "Samsung NX1100", 0, 0,
8123.        { 6933,-2268,-753,-4921,13387,1647,-803,1641,6096 } },
8124.    { "Samsung NX11", 0, 0,
8125.        { 10332,-3234,-1168,-6111,14639,1520,-1352,2647,8331 } },
8126.    { "Samsung NX10", 0, 0,     /* also NX100 */
8127.        { 10332,-3234,-1168,-6111,14639,1520,-1352,2647,8331 } },
8128.    { "Samsung NX500", 0, 0,
8129.        { 10686,-4042,-1052,-3595,13238,276,-464,1259,5931 } },
8130.    { "Samsung NX5", 0, 0,
8131.        { 10332,-3234,-1168,-6111,14639,1520,-1352,2647,8331 } },
8132.    { "Samsung NX1", 0, 0,
8133.        { 10686,-4042,-1052,-3595,13238,276,-464,1259,5931 } },
8134.    { "Samsung WB2000", 0, 0xfff,
8135.        { 12093,-3557,-1155,-1000,9534,1733,-22,1787,4576 } },
8136.    { "Samsung GX-1", 0, 0,
8137.        { 10504,-2438,-1189,-8603,16207,2531,-1022,863,12242 } },
8138.    { "Samsung GX20", 0, 0,     /* copied from Pentax K20D */
8139.        { 9427,-2714,-868,-7493,16092,1373,-2199,3264,7180 } },
8140.    { "Samsung S85", 0, 0,              /* DJC */
8141.        { 11885,-3968,-1473,-4214,12299,1916,-835,1655,5549 } },
8142.    { "Sinar", 0, 0,                    /* DJC */
8143.        { 16442,-2956,-2422,-2877,12128,750,-1136,6066,4559 } },
8144.    { "Sony DSC-F828", 0, 0,
8145.        { 7924,-1910,-777,-8226,15459,2998,-1517,2199,6818,-7242,11401,3481 } },
8146.    { "Sony DSC-R1", 0, 0,
8147.        { 8512,-2641,-694,-8042,15670,2526,-1821,2117,7414 } },
8148.    { "Sony DSC-V3", 0, 0,
8149.        { 7511,-2571,-692,-7894,15088,3060,-948,1111,8128 } },
8150.    { "Sony DSC-RX100M", 0, 0,          /* M2, M3, M4, and M5 */
8151.        { 6596,-2079,-562,-4782,13016,1933,-970,1581,5181 } },
8152.    { "Sony DSC-RX100", 0, 0,
8153.        { 8651,-2754,-1057,-3464,12207,1373,-568,1398,4434 } },
8154.    { "Sony DSC-RX10M4", 0, 0,
8155.        { 7699,-2566,-629,-2967,11270,1928,-378,1286,4807 } },
8156.    { "Sony DSC-RX10", 0, 0,            /* also RX10M2, RX10M3 */
8157.        { 6679,-1825,-745,-5047,13256,1953,-1580,2422,5183 } },
8158.    { "Sony DSC-RX1RM2", 0, 0,
8159.        { 6629,-1900,-483,-4618,12349,2550,-622,1381,6514 } },
8160.    { "Sony DSC-RX1", 0, 0,
8161.        { 6344,-1612,-462,-4863,12477,2681,-865,1786,6899 } },
8162.    { "Sony DSC-RX0", 200, 0,
8163.        { 9396,-3507,-843,-2497,11111,1572,-343,1355,5089 } },
8164.    { "Sony DSLR-A100", 0, 0xfeb,
8165.        { 9437,-2811,-774,-8405,16215,2290,-710,596,7181 } },
8166.    { "Sony DSLR-A290", 0, 0,
8167.        { 6038,-1484,-579,-9145,16746,2512,-875,746,7218 } },
8168.    { "Sony DSLR-A2", 0, 0,
8169.        { 9847,-3091,-928,-8485,16345,2225,-715,595,7103 } },
8170.    { "Sony DSLR-A300", 0, 0,
8171.        { 9847,-3091,-928,-8485,16345,2225,-715,595,7103 } },
8172.    { "Sony DSLR-A330", 0, 0,
8173.        { 9847,-3091,-929,-8485,16346,2225,-714,595,7103 } },
8174.    { "Sony DSLR-A350", 0, 0xffc,
8175.        { 6038,-1484,-578,-9146,16746,2513,-875,746,7217 } },
8176.    { "Sony DSLR-A380", 0, 0,
8177.        { 6038,-1484,-579,-9145,16746,2512,-875,746,7218 } },
8178.    { "Sony DSLR-A390", 0, 0,
8179.        { 6038,-1484,-579,-9145,16746,2512,-875,746,7218 } },
8180.    { "Sony DSLR-A450", 0, 0xfeb,
8181.        { 4950,-580,-103,-5228,12542,3029,-709,1435,7371 } },
8182.    { "Sony DSLR-A580", 0, 0xfeb,
8183.        { 5932,-1492,-411,-4813,12285,2856,-741,1524,6739 } },
```

```
8184.     { "Sony DSLR-A500", 0, 0xfeb,
8185.         { 6046,-1127,-278,-5574,13076,2786,-691,1419,7625 } },
8186.     { "Sony DSLR-A5", 0, 0xfeb,
8187.         { 4950,-580,-103,-5228,12542,3029,-709,1435,7371 } },
8188.     { "Sony DSLR-A700", 0, 0,
8189.         { 5775,-805,-359,-8574,16295,2391,-1943,2341,7249 } },
8190.     { "Sony DSLR-A850", 0, 0,
8191.         { 5413,-1162,-365,-5665,13098,2866,-608,1179,8440 } },
8192.     { "Sony DSLR-A900", 0, 0,
8193.         { 5209,-1072,-397,-8845,16120,2919,-1618,1803,8654 } },
8194.     { "Sony ILCA-68", 0, 0,
8195.         { 6435,-1903,-536,-4722,12449,2550,-663,1363,6517 } },
8196.     { "Sony ILCA-77M2", 0, 0,
8197.         { 5991,-1732,-443,-4100,11989,2381,-704,1467,5992 } },
8198.     { "Sony ILCA-99M2", 0, 0,
8199.         { 6660,-1918,-471,-4613,12398,2485,-649,1433,6447 } },
8200.     { "Sony ILCE-6", 0, 0,                  /* 6300, 6500 */
8201.         { 5973,-1695,-419,-3826,11797,2293,-639,1398,5789 } },
8202.     { "Sony ILCE-7M2", 0, 0,
8203.         { 5271,-712,-347,-6153,13653,2763,-1601,2366,7242 } },
8204.     { "Sony ILCE-7M3", 0, 0,
8205.         { 7374,-2389,-551,-5435,13162,2519,-1006,1795,6552 } },
8206.     { "Sony ILCE-7S", 0, 0,     /* also ILCE-7SM2 */
8207.         { 5838,-1430,-246,-3497,11477,2297,-748,1885,5778 } },
8208.     { "Sony ILCE-7RM3", 0, 0,
8209.         { 6640,-1847,-503,-5238,13010,2474,-993,1673,6527 } },
8210.     { "Sony ILCE-7RM2", 0, 0,
8211.         { 6629,-1900,-483,-4618,12349,2550,-622,1381,6514 } },
8212.     { "Sony ILCE-7R", 0, 0,
8213.         { 4913,-541,-202,-6130,13513,2906,-1564,2151,7183 } },
8214.     { "Sony ILCE-7", 0, 0,
8215.         { 5271,-712,-347,-6153,13653,2763,-1601,2366,7242 } },
8216.     { "Sony ILCE-9", 0, 0,
8217.         { 6389,-1703,-378,-4562,12265,2587,-670,1489,6550 } },
8218.     { "Sony ILCE", 0, 0,        /* 3000, 5000, 5100, 6000, and QX1 */
8219.         { 5991,-1456,-455,-4764,12135,2980,-707,1425,6701 } },
8220.     { "Sony NEX-5N", 0, 0,
8221.         { 5991,-1456,-455,-4764,12135,2980,-707,1425,6701 } },
8222.     { "Sony NEX-5R", 0, 0,
8223.         { 6129,-1545,-418,-4930,12490,2743,-977,1693,6615 } },
8224.     { "Sony NEX-5T", 0, 0,
8225.         { 6129,-1545,-418,-4930,12490,2743,-977,1693,6615 } },
8226.     { "Sony NEX-3N", 0, 0,
8227.         { 6129,-1545,-418,-4930,12490,2743,-977,1693,6615 } },
8228.     { "Sony NEX-3", 138, 0,             /* DJC */
8229.         { 6907,-1256,-645,-4940,12621,2320,-1710,2581,6230 } },
8230.     { "Sony NEX-5", 116, 0,             /* DJC */
8231.         { 6807,-1350,-342,-4216,11649,2567,-1089,2001,6420 } },
8232.     { "Sony NEX-3", 0, 0,               /* Adobe */
8233.         { 6549,-1550,-436,-4880,12435,2753,-854,1868,6976 } },
8234.     { "Sony NEX-5", 0, 0,               /* Adobe */
8235.         { 6549,-1550,-436,-4880,12435,2753,-854,1868,6976 } },
8236.     { "Sony NEX-6", 0, 0,
8237.         { 6129,-1545,-418,-4930,12490,2743,-977,1693,6615 } },
8238.     { "Sony NEX-7", 0, 0,
8239.         { 5491,-1192,-363,-4951,12342,2948,-911,1722,7192 } },
8240.     { "Sony NEX", 0, 0, /* NEX-C3, NEX-F3 */
8241.         { 5991,-1456,-455,-4764,12135,2980,-707,1425,6701 } },
8242.     { "Sony SLT-A33", 0, 0,
8243.         { 6069,-1221,-366,-5221,12779,2734,-1024,2066,6834 } },
8244.     { "Sony SLT-A35", 0, 0,
8245.         { 5986,-1618,-415,-4557,11820,3120,-681,1404,6971 } },
8246.     { "Sony SLT-A37", 0, 0,
8247.         { 5991,-1456,-455,-4764,12135,2980,-707,1425,6701 } },
8248.     { "Sony SLT-A55", 0, 0,
```

```
        { 5932,-1492,-411,-4813,12285,2856,-741,1524,6739 } },
    { "Sony SLT-A57", 0, 0,
        { 5991,-1456,-455,-4764,12135,2980,-707,1425,6701 } },
    { "Sony SLT-A58", 0, 0,
        { 5991,-1456,-455,-4764,12135,2980,-707,1425,6701 } },
    { "Sony SLT-A65", 0, 0,
        { 5491,-1192,-363,-4951,12342,2948,-911,1722,7192 } },
    { "Sony SLT-A77", 0, 0,
        { 5491,-1192,-363,-4951,12342,2948,-911,1722,7192 } },
    { "Sony SLT-A99", 0, 0,
        { 6344,-1612,-462,-4863,12477,2681,-865,1786,6899 } },
    { "YI M1", 0, 0,
        { 7712,-2059,-653,-3882,11494,2726,-710,1332,5958 } },
  };
  double cam_xyz[4][3];
  char name[130];
  int i, j;

  sprintf (name, "%s %s", make, model);
  for (i=0; i < sizeof table / sizeof *table; i++)
    if (!strncmp (name, table[i].prefix, strlen(table[i].prefix))) {
      if (table[i].black)   black   = (ushort) table[i].black;
      if (table[i].maximum) maximum = (ushort) table[i].maximum;
      if (table[i].trans[0]) {
        for (raw_color = j=0; j < 12; j++)
          ((double *)cam_xyz)[j] = table[i].trans[j] / 10000.0;
        cam_xyz_coeff (rgb_cam, cam_xyz);
      }
      break;
    }
}

void CLASS simple_coeff (int index)
{
  static const float table[][12] = {
  /* index 0 -- all Foveon cameras */
  { 1.4032,-0.2231,-0.1016,-0.5263,1.4816,0.017,-0.0112,0.0183,0.9113 },
  /* index 1 -- Kodak DC20 and DC25 */
  { 2.25,0.75,-1.75,-0.25,-0.25,0.75,0.75,-0.25,-0.25,-1.75,0.75,2.25 },
  /* index 2 -- Logitech Fotoman Pixtura */
  { 1.893,-0.418,-0.476,-0.495,1.773,-0.278,-1.017,-0.655,2.672 },
  /* index 3 -- Nikon E880, E900, and E990 */
  { -1.936280,  1.800443, -1.448486,  2.584324,
     1.405365, -0.524955, -0.289090,  0.408680,
    -1.204965,  1.082304,  2.941367, -1.818705 }
  };
  int i, c;

  for (raw_color = i=0; i < 3; i++)
    FORCC rgb_cam[i][c] = table[index][i*colors+c];
}

short CLASS guess_byte_order (int words)
{
  uchar test[4][2];
  int t=2, msb;
  double diff, sum[2] = {0,0};

  fread (test[0], 2, 2, ifp);
  for (words-=2; words--; ) {
    fread (test[t], 2, 1, ifp);
    for (msb=0; msb < 2; msb++) {
      diff = (test[t^2][msb] << 8 | test[t^2][!msb])
           - (test[t  ][msb] << 8 | test[t  ][!msb]);
      sum[msb] += diff*diff;
```

```
    }
    t = (t+1) & 3;
  }
  return sum[0] < sum[1] ? 0x4d4d : 0x4949;
}

float CLASS find_green (int bps, int bite, int off0, int off1)
{
  UINT64 bitbuf=0;
  int vbits, col, i, c;
  ushort img[2][2064];
  double sum[]={0,0};

  FORC(2) {
    fseek (ifp, c ? off1:off0, SEEK_SET);
    for (vbits=col=0; col < width; col++) {
      for (vbits -= bps; vbits < 0; vbits += bite) {
        bitbuf <<= bite;
        for (i=0; i < bite; i+=8)
          bitbuf |= (unsigned) (fgetc(ifp) << i);
      }
      img[c][col] = bitbuf << (64-bps-vbits) >> (64-bps);
    }
  }
  FORC(width-1) {
    sum[ c & 1] += ABS(img[0][c]-img[1][c+1]);
    sum[~c & 1] += ABS(img[1][c]-img[0][c+1]);
  }
  return 100 * log(sum[0]/sum[1]);
}

/*
   Identify which camera created this file, and set global variables
   accordingly.
 */
void CLASS identify()
{
  static const short pana[][6] = {
    { 3130, 1743,  4,  0, -6,  0 },
    { 3130, 2055,  4,  0, -6,  0 },
    { 3130, 2319,  4,  0, -6,  0 },
    { 3170, 2103, 18,  0,-42, 20 },
    { 3170, 2367, 18, 13,-42,-21 },
    { 3177, 2367,  0,  0, -1,  0 },
    { 3304, 2458,  0,  0, -1,  0 },
    { 3330, 2463,  9,  0, -5,  0 },
    { 3330, 2479,  9,  0,-17,  4 },
    { 3370, 1899, 15,  0,-44, 20 },
    { 3370, 2235, 15,  0,-44, 20 },
    { 3370, 2511, 15, 10,-44,-21 },
    { 3690, 2751,  3,  0, -8, -3 },
    { 3710, 2751,  0,  0, -3,  0 },
    { 3724, 2450,  0,  0,  0, -2 },
    { 3770, 2487, 17,  0,-44, 19 },
    { 3770, 2799, 17, 15,-44,-19 },
    { 3880, 2170,  6,  0, -6,  0 },
    { 4060, 3018,  0,  0,  0, -2 },
    { 4290, 2391,  3,  0, -8, -1 },
    { 4330, 2439, 17, 15,-44,-19 },
    { 4508, 2962,  0,  0, -3, -4 },
    { 4508, 3330,  0,  0, -3, -6 },
  };
  static const ushort canon[][11] = {
    { 1944, 1416,   0,  0, 48,  0 },
    { 2144, 1560,   4,  8, 52,  2, 0, 0, 0, 25 },
```

```
8379.    { 2224, 1456,  48,  6,  0,  2 },
8380.    { 2376, 1728,  12,  6, 52,  2 },
8381.    { 2672, 1968,  12,  6, 44,  2 },
8382.    { 3152, 2068,  64, 12,  0,  0, 16 },
8383.    { 3160, 2344,  44, 12,  4,  4 },
8384.    { 3344, 2484,   4,  6, 52,  6 },
8385.    { 3516, 2328,  42, 14,  0,  0 },
8386.    { 3596, 2360,  74, 12,  0,  0 },
8387.    { 3744, 2784,  52, 12,  8, 12 },
8388.    { 3944, 2622,  30, 18,  6,  2 },
8389.    { 3948, 2622,  42, 18,  0,  2 },
8390.    { 3984, 2622,  76, 20,  0,  2, 14 },
8391.    { 4104, 3048,  48, 12, 24, 12 },
8392.    { 4116, 2178,   4,  2,  0,  0 },
8393.    { 4152, 2772, 192, 12,  0,  0 },
8394.    { 4160, 3124, 104, 11,  8, 65 },
8395.    { 4176, 3062,  96, 17,  8,  0, 0, 16, 0, 7, 0x49 },
8396.    { 4192, 3062,  96, 17, 24,  0, 0, 16, 0, 0, 0x49 },
8397.    { 4312, 2876,  22, 18,  0,  2 },
8398.    { 4352, 2874,  62, 18,  0,  0 },
8399.    { 4476, 2954,  90, 34,  0,  0 },
8400.    { 4480, 3348,  12, 10, 36, 12, 0, 0, 0, 18, 0x49 },
8401.    { 4480, 3366,  80, 50,  0,  0 },
8402.    { 4496, 3366,  80, 50, 12,  0 },
8403.    { 4768, 3516,  96, 16,  0,  0, 0, 16 },
8404.    { 4832, 3204,  62, 26,  0,  0 },
8405.    { 4832, 3228,  62, 51,  0,  0 },
8406.    { 5108, 3349,  98, 13,  0,  0 },
8407.    { 5120, 3318, 142, 45, 62,  0 },
8408.    { 5280, 3528,  72, 52,  0,  0 },
8409.    { 5344, 3516, 142, 51,  0,  0 },
8410.    { 5344, 3584, 126,100,  0,  2 },
8411.    { 5360, 3516, 158, 51,  0,  0 },
8412.    { 5568, 3708,  72, 38,  0,  0 },
8413.    { 5632, 3710,  96, 17,  0,  0, 0, 16, 0, 0, 0x49 },
8414.    { 5712, 3774,  62, 20, 10,  2 },
8415.    { 5792, 3804, 158, 51,  0,  0 },
8416.    { 5920, 3950, 122, 80,  2,  0 },
8417.    { 6096, 4051,  76, 35,  0,  0 },
8418.    { 6096, 4056,  72, 34,  0,  0 },
8419.    { 6288, 4056, 264, 36,  0,  0 },
8420.    { 6384, 4224, 120, 44,  0,  0 },
8421.    { 6880, 4544, 136, 42,  0,  0 },
8422.    { 8896, 5920, 160, 64,  0,  0 },
8423.  };
8424.  static const struct {
8425.    ushort id;
8426.    char model[20];
8427.  } unique[] = {
8428.    { 0x168, "EOS 10D" },    { 0x001, "EOS-1D" },
8429.    { 0x175, "EOS 20D" },    { 0x174, "EOS-1D Mark II" },
8430.    { 0x234, "EOS 30D" },    { 0x232, "EOS-1D Mark II N" },
8431.    { 0x190, "EOS 40D" },    { 0x169, "EOS-1D Mark III" },
8432.    { 0x261, "EOS 50D" },    { 0x281, "EOS-1D Mark IV" },
8433.    { 0x287, "EOS 60D" },    { 0x167, "EOS-1DS" },
8434.    { 0x325, "EOS 70D" },
8435.    { 0x408, "EOS 77D" },    { 0x331, "EOS M" },
8436.    { 0x350, "EOS 80D" },    { 0x328, "EOS-1D X Mark II" },
8437.    { 0x346, "EOS 100D" },
8438.    { 0x417, "EOS 200D" },
8439.    { 0x170, "EOS 300D" },   { 0x188, "EOS-1Ds Mark II" },
8440.    { 0x176, "EOS 450D" },   { 0x215, "EOS-1Ds Mark III" },
8441.    { 0x189, "EOS 350D" },   { 0x324, "EOS-1D C" },
8442.    { 0x236, "EOS 400D" },   { 0x269, "EOS-1D X" },
8443.    { 0x252, "EOS 500D" },   { 0x213, "EOS 5D" },
```

```
8444.    { 0x270, "EOS 550D" },   { 0x218, "EOS 5D Mark II" },
8445.    { 0x286, "EOS 600D" },   { 0x285, "EOS 5D Mark III" },
8446.    { 0x301, "EOS 650D" },   { 0x302, "EOS 6D" },
8447.    { 0x326, "EOS 700D" },   { 0x250, "EOS 7D" },
8448.    { 0x393, "EOS 750D" },   { 0x289, "EOS 7D Mark II" },
8449.    { 0x347, "EOS 760D" },   { 0x406, "EOS 6D Mark II" },
8450.    { 0x405, "EOS 800D" },   { 0x349, "EOS 5D Mark IV" },
8451.    { 0x254, "EOS 1000D" },
8452.    { 0x288, "EOS 1100D" },
8453.    { 0x327, "EOS 1200D" },  { 0x382, "EOS 5DS" },
8454.    { 0x404, "EOS 1300D" },  { 0x401, "EOS 5DS R" },
8455.    { 0x422, "EOS 1500D" },
8456.    { 0x432, "EOS 3000D" },
8457.  }, sonique[] = {
8458.    { 0x002, "DSC-R1" },     { 0x100, "DSLR-A100" },
8459.    { 0x101, "DSLR-A900" },  { 0x102, "DSLR-A700" },
8460.    { 0x103, "DSLR-A200" },  { 0x104, "DSLR-A350" },
8461.    { 0x105, "DSLR-A300" },  { 0x108, "DSLR-A330" },
8462.    { 0x109, "DSLR-A230" },  { 0x10a, "DSLR-A290" },
8463.    { 0x10d, "DSLR-A850" },  { 0x111, "DSLR-A550" },
8464.    { 0x112, "DSLR-A500" },  { 0x113, "DSLR-A450" },
8465.    { 0x116, "NEX-5" },      { 0x117, "NEX-3" },
8466.    { 0x118, "SLT-A33" },    { 0x119, "SLT-A55V" },
8467.    { 0x11a, "DSLR-A560" },  { 0x11b, "DSLR-A580" },
8468.    { 0x11c, "NEX-C3" },     { 0x11d, "SLT-A35" },
8469.    { 0x11e, "SLT-A65V" },   { 0x11f, "SLT-A77V" },
8470.    { 0x120, "NEX-5N" },     { 0x121, "NEX-7" },
8471.    { 0x123, "SLT-A37" },    { 0x124, "SLT-A57" },
8472.    { 0x125, "NEX-F3" },     { 0x126, "SLT-A99V" },
8473.    { 0x127, "NEX-6" },      { 0x128, "NEX-5R" },
8474.    { 0x129, "DSC-RX100" },  { 0x12a, "DSC-RX1" },
8475.    { 0x12e, "ILCE-3000" },  { 0x12f, "SLT-A58" },
8476.    { 0x131, "NEX-3N" },     { 0x132, "ILCE-7" },
8477.    { 0x133, "NEX-5T" },     { 0x134, "DSC-RX100M2" },
8478.    { 0x135, "DSC-RX10" },   { 0x136, "DSC-RX1R" },
8479.    { 0x137, "ILCE-7R" },    { 0x138, "ILCE-6000" },
8480.    { 0x139, "ILCE-5000" },  { 0x13d, "DSC-RX100M3" },
8481.    { 0x13e, "ILCE-7S" },    { 0x13f, "ILCA-77M2" },
8482.    { 0x153, "ILCE-5100" },  { 0x154, "ILCE-7M2" },
8483.    { 0x155, "DSC-RX100M4" },{ 0x156, "DSC-RX10M2" },
8484.    { 0x158, "DSC-RX1RM2" }, { 0x15a, "ILCE-QX1" },
8485.    { 0x15b, "ILCE-7RM2" },  { 0x15e, "ILCE-7SM2" },
8486.    { 0x161, "ILCA-68" },    { 0x162, "ILCA-99M2" },
8487.    { 0x163, "DSC-RX10M3" }, { 0x164, "DSC-RX100M5" },
8488.    { 0x165, "ILCE-6300" },  { 0x166, "ILCE-9" },
8489.    { 0x168, "ILCE-6500" },  { 0x16a, "ILCE-7RM3" },
8490.    { 0x16b, "ILCE-7M3" },   { 0x16c, "DSC-RX0" },
8491.    { 0x16d, "DSC-RX10M4" },
8492.  };
8493.  static const char *orig, panalias[][12] = {
8494.    "@DC-FZ80", "DC-FZ82", "DC-FZ85",
8495.    "@DC-FZ81", "DC-FZ83",
8496.    "@DC-GF9", "DC-GX800", "DC-GX850",
8497.    "@DC-GF10", "DC-GF90",
8498.    "@DC-GX9", "DC-GX7MK3",
8499.    "@DC-ZS70", "DC-TZ90", "DC-TZ91", "DC-TZ92", "DC-TZ93",
8500.    "@DMC-FZ40", "DMC-FZ45",
8501.    "@DMC-FZ2500", "DMC-FZ2000", "DMC-FZH1",
8502.    "@DMC-G8", "DMC-G80", "DMC-G81", "DMC-G85",
8503.    "@DMC-GX85", "DMC-GX80", "DMC-GX7MK2",
8504.    "@DMC-LX9", "DMC-LX10", "DMC-LX15",
8505.    "@DMC-ZS40", "DMC-TZ60", "DMC-TZ61",
8506.    "@DMC-ZS50", "DMC-TZ70", "DMC-TZ71",
8507.    "@DMC-ZS60", "DMC-TZ80", "DMC-TZ81", "DMC-TZ85",
8508.    "@DMC-ZS100", "DMC-ZS110", "DMC-TZ100", "DMC-TZ101", "DMC-TZ110", "DMC-TX1",
```

```
8509.    "@DC-ZS200", "DC-TX2", "DC-TZ200", "DC-TZ202", "DC-TZ220", "DC-ZS220",
8510.  };
8511.  static const struct {
8512.    unsigned fsize;
8513.    ushort rw, rh;
8514.    uchar lm, tm, rm, bm, lf, cf, max, flags;
8515.    char make[10], model[20];
8516.    ushort offset;
8517.  } table[] = {
8518.    {   786432,1024, 768, 0, 0, 0, 0, 0,0x94,0,0,"AVT","F-080C" },
8519.    {  1447680,1392,1040, 0, 0, 0, 0, 0,0x94,0,0,"AVT","F-145C" },
8520.    {  1920000,1600,1200, 0, 0, 0, 0, 0,0x94,0,0,"AVT","F-201C" },
8521.    {  5067304,2588,1958, 0, 0, 0, 0, 0,0x94,0,0,"AVT","F-510C" },
8522.    {  5067316,2588,1958, 0, 0, 0, 0, 0,0x94,0,0,"AVT","F-510C",12 },
8523.    { 10134608,2588,1958, 0, 0, 0, 0, 9,0x94,0,0,"AVT","F-510C" },
8524.    { 10134620,2588,1958, 0, 0, 0, 0, 9,0x94,0,0,"AVT","F-510C",12 },
8525.    { 16157136,3272,2469, 0, 0, 0, 0, 9,0x94,0,0,"AVT","F-810C" },
8526.    { 15980544,3264,2448, 0, 0, 0, 0, 8,0x61,0,1,"AgfaPhoto","DC-833m" },
8527.    {  9631728,2532,1902, 0, 0, 0, 0,96,0x61,0,0,"Alcatel","5035D" },
8528.    {  2868726,1384,1036, 0, 0, 0, 0,64,0x49,0,8,"Baumer","TXG14",1078 },
8529.    {  5298000,2400,1766,12,12,44, 2, 8,0x94,0,2,"Canon","PowerShot SD300" },
8530.    {  6553440,2664,1968, 4, 4,44, 4, 8,0x94,0,2,"Canon","PowerShot A460" },
8531.    {  6573120,2672,1968,12, 8,44, 0, 8,0x94,0,2,"Canon","PowerShot A610" },
8532.    {  6653280,2672,1992,10, 6,42, 2, 8,0x94,0,2,"Canon","PowerShot A530" },
8533.    {  7710960,2888,2136,44, 8, 4, 0, 8,0x94,0,2,"Canon","PowerShot S3 IS" },
8534.    {  9219600,3152,2340,36,12, 4, 0, 8,0x94,0,2,"Canon","PowerShot A620" },
8535.    {  9243240,3152,2346,12, 7,44,13, 8,0x49,0,2,"Canon","PowerShot A470" },
8536.    { 10341600,3336,2480, 6, 5,32, 3, 8,0x94,0,2,"Canon","PowerShot A720 IS" },
8537.    { 10383120,3344,2484,12, 6,44, 6, 8,0x94,0,2,"Canon","PowerShot A630" },
8538.    { 12945240,3736,2772,12, 6,52, 6, 8,0x94,0,2,"Canon","PowerShot A640" },
8539.    { 15636240,4104,3048,48,12,24,12, 8,0x94,0,2,"Canon","PowerShot A650" },
8540.    { 15467760,3720,2772, 6,12,30, 0, 8,0x94,0,2,"Canon","PowerShot SX110 IS" },
8541.    { 15534576,3728,2778,12, 9,44, 9, 8,0x94,0,2,"Canon","PowerShot SX120 IS" },
8542.    { 18653760,4080,3048,24,12,24,12, 8,0x94,0,2,"Canon","PowerShot SX20 IS" },
8543.    { 19131120,4168,3060,92,16, 4, 1, 8,0x94,0,2,"Canon","PowerShot SX220 HS" },
8544.    { 21936096,4464,3276,25,10,73,12, 8,0x16,0,2,"Canon","PowerShot SX30 IS" },
8545.    { 24724224,4704,3504, 8,16,56, 8, 8,0x94,0,2,"Canon","PowerShot A3300 IS" },
8546.    { 30858240,5248,3920, 8,16,56,16, 8,0x94,0,2,"Canon","IXUS 160" },
8547.    {  1976352,1632,1211, 0, 2, 0, 1, 0,0x94,0,1,"Casio","QV-2000UX" },
8548.    {  3217760,2080,1547, 0, 0,10, 1, 0,0x94,0,1,"Casio","QV-3*00EX" },
8549.    {  6218368,2585,1924, 0, 0, 9, 0, 0,0x94,0,1,"Casio","QV-5700" },
8550.    {  7816704,2867,2181, 0, 0,34,36, 0,0x16,0,1,"Casio","EX-Z60" },
8551.    {  2937856,1621,1208, 0, 0, 1, 0, 0,0x94,7,13,"Casio","EX-S20" },
8552.    {  4948608,2090,1578, 0, 0,32,34, 0,0x94,7,1,"Casio","EX-S100" },
8553.    {  6054400,2346,1720, 2, 0,32, 0, 0,0x94,7,1,"Casio","QV-R41" },
8554.    {  7426656,2568,1928, 0, 0, 0, 0, 0,0x94,0,1,"Casio","EX-P505" },
8555.    {  7530816,2602,1929, 0, 0,22, 0, 0,0x94,7,1,"Casio","QV-R51" },
8556.    {  7542528,2602,1932, 0, 0,32, 0, 0,0x94,7,1,"Casio","EX-Z50" },
8557.    {  7562048,2602,1937, 0, 0,25, 0, 0,0x16,7,1,"Casio","EX-Z500" },
8558.    {  7753344,2602,1986, 0, 0,32,26, 0,0x94,7,1,"Casio","EX-Z55" },
8559.    {  9313536,2858,2172, 0, 0,14,30, 0,0x94,7,1,"Casio","EX-P600" },
8560.    { 10834368,3114,2319, 0, 0,27, 0, 0,0x94,0,1,"Casio","EX-Z750" },
8561.    { 10843712,3114,2321, 0, 0,25, 0, 0,0x94,0,1,"Casio","EX-Z75" },
8562.    { 10979200,3114,2350, 0, 0,32,32, 0,0x94,7,1,"Casio","EX-P700" },
8563.    { 12310144,3285,2498, 0, 0, 6,30, 0,0x94,0,1,"Casio","EX-Z850" },
8564.    { 12489984,3328,2502, 0, 0,47,35, 0,0x94,0,1,"Casio","EX-Z8" },
8565.    { 15499264,3754,2752, 0, 0,82, 0, 0,0x94,0,1,"Casio","EX-Z1050" },
8566.    { 18702336,4096,3044, 0, 0,24, 0,80,0x94,7,1,"Casio","EX-ZR100" },
8567.    {  7684000,2260,1700, 0, 0, 0, 0,13,0x94,0,1,"Casio","QV-4000" },
8568.    {   787456,1024, 769, 0, 1, 0, 0, 0,0x49,0,0,"Creative","PC-CAM 600" },
8569.    { 28829184,4384,3288, 0, 0, 0, 0,36,0x61,0,0,"DJI" },
8570.    { 15151104,4608,3288, 0, 0, 0, 0, 0,0x94,0,0,"Matrix" },
8571.    {  3840000,1600,1200, 0, 0, 0, 0,65,0x49,0,0,"Foculus","531C" },
8572.    {   307200, 640, 480, 0, 0, 0, 0, 0,0x94,0,0,"Generic" },
8573.    {    62464, 256, 244, 1, 1, 6, 1, 0,0x8d,0,0,"Kodak","DC20" },
```

```
8574.    {   124928, 512, 244, 1, 1,10, 1, 0,0x8d,0,0,"Kodak","DC20" },
8575.    {  1652736,1536,1076, 0,52, 0, 0, 0,0x61,0,0,"Kodak","DCS200" },
8576.    {  4159302,2338,1779, 1,33, 1, 2, 0,0x94,0,0,"Kodak","C330" },
8577.    {  4162462,2338,1779, 1,33, 1, 2, 0,0x94,0,0,"Kodak","C330",3160 },
8578.    {  2247168,1232, 912, 0, 0,16, 0, 0,0x00,0,0,"Kodak","C330" },
8579.    {  3370752,1232, 912, 0, 0,16, 0, 0,0x00,0,0,"Kodak","C330" },
8580.    {  6163328,2864,2152, 0, 0, 0, 0, 0,0x94,0,0,"Kodak","C603" },
8581.    {  6166488,2864,2152, 0, 0, 0, 0, 0,0x94,0,0,"Kodak","C603",3160 },
8582.    {   460800, 640, 480, 0, 0, 0, 0, 0,0x00,0,0,"Kodak","C603" },
8583.    {  9116448,2848,2134, 0, 0, 0, 0, 0,0x00,0,0,"Kodak","C603" },
8584.    { 12241200,4040,3030, 2, 0, 0,13, 0,0x49,0,0,"Kodak","12MP" },
8585.    { 12272756,4040,3030, 2, 0, 0,13, 0,0x49,0,0,"Kodak","12MP",31556 },
8586.    { 18000000,4000,3000, 0, 0, 0, 0, 0,0x00,0,0,"Kodak","12MP" },
8587.    {   614400, 640, 480, 0, 3, 0, 0,64,0x94,0,0,"Kodak","KAI-0340" },
8588.    { 15360000,3200,2400, 0, 0, 0, 0,96,0x16,0,0,"Lenovo","A820" },
8589.    {  3884928,1608,1207, 0, 0, 0, 0,96,0x16,0,0,"Micron","2010",3212 },
8590.    {  1138688,1534, 986, 0, 0, 0, 0, 0,0x61,0,0,"Minolta","RD175",513 },
8591.    {  1581060,1305, 969, 0, 0,18, 6, 6,0x1e,4,1,"Nikon","E900" },
8592.    {  2465792,1638,1204, 0, 0,22, 1, 6,0x4b,5,1,"Nikon","E950" },
8593.    {  2940928,1616,1213, 0, 0, 0, 7,30,0x94,0,1,"Nikon","E2100" },
8594.    {  4771840,2064,1541, 0, 0, 0, 1, 6,0xe1,0,1,"Nikon","E990" },
8595.    {  4775936,2064,1542, 0, 0, 0, 0,30,0x94,0,1,"Nikon","E3700" },
8596.    {  5865472,2288,1709, 0, 0, 0, 1, 6,0xb4,0,1,"Nikon","E4500" },
8597.    {  5869568,2288,1710, 0, 0, 0, 0, 6,0x16,0,1,"Nikon","E4300" },
8598.    {  7438336,2576,1925, 0, 0, 0, 1, 6,0xb4,0,1,"Nikon","E5000" },
8599.    {  8998912,2832,2118, 0, 0, 0, 0,30,0x94,7,1,"Nikon","COOLPIX S6" },
8600.    {  5939200,2304,1718, 0, 0, 0, 0,30,0x16,0,0,"Olympus","C770UZ" },
8601.    {  3178560,2064,1540, 0, 0, 0, 0, 0,0x94,0,1,"Pentax","Optio S" },
8602.    {  4841984,2090,1544, 0, 0,22, 0, 0,0x94,7,1,"Pentax","Optio S" },
8603.    {  6114240,2346,1737, 0, 0,22, 0, 0,0x94,7,1,"Pentax","Optio S4" },
8604.    { 10702848,3072,2322, 0, 0, 0,21,30,0x94,0,1,"Pentax","Optio 750Z" },
8605.    {  4147200,1920,1080, 0, 0, 0, 0, 0,0x49,0,0,"Photron","BC2-HD" },
8606.    {  4151666,1920,1080, 0, 0, 0, 0, 0,0x49,0,0,"Photron","BC2-HD",8 },
8607.    { 13248000,2208,3000, 0, 0, 0, 0,13,0x61,0,0,"Pixelink","A782" },
8608.    {  6291456,2048,1536, 0, 0, 0, 0,96,0x61,0,0,"RoverShot","3320AF" },
8609.    {   311696, 644, 484, 0, 0, 0, 0, 0,0x16,0,8,"ST Micro","STV680 VGA" },
8610.    { 16098048,3288,2448, 0, 0,24, 0, 9,0x94,0,1,"Samsung","S85" },
8611.    { 16215552,3312,2448, 0, 0,48, 0, 9,0x94,0,1,"Samsung","S85" },
8612.    { 20487168,3648,2808, 0, 0, 0, 0,13,0x94,5,1,"Samsung","WB550" },
8613.    { 24000000,4000,3000, 0, 0, 0, 0,13,0x94,5,1,"Samsung","WB550" },
8614.    { 12582980,3072,2048, 0, 0, 0, 0,33,0x61,0,0,"Sinar","",68 },
8615.    { 33292868,4080,4080, 0, 0, 0, 0,33,0x61,0,0,"Sinar","",68 },
8616.    { 44390468,4080,5440, 0, 0, 0, 0,33,0x61,0,0,"Sinar","",68 },
8617.    {  1409024,1376,1024, 0, 0, 1, 0, 0,0x49,0,0,"Sony","XCD-SX910CR" },
8618.    {  2818048,1376,1024, 0, 0, 1, 0,97,0x49,0,0,"Sony","XCD-SX910CR" },
8619.    { 17496000,4320,3240, 0, 0, 0,0,224,0x94,0,0,"Xiro","Xplorer V" },
8620.  };
8621.  static const char *corp[] =
8622.    { "AgfaPhoto", "Canon", "Casio", "Epson", "Fujifilm",
8623.      "Mamiya", "Minolta", "Motorola", "Kodak", "Konica", "Leica",
8624.      "Nikon", "Nokia", "Olympus", "Ricoh", "Pentax", "Phase One",
8625.      "Samsung", "Sigma", "Sinar", "Sony", "YI" };
8626.  char head[32], *cp;
8627.  int hlen, flen, fsize, zero_fsize=1, i, c;
8628.  struct jhead jh;
8629.
8630.  tiff_flip = flip = filters = UINT_MAX;        /* unknown */
8631.  raw_height = raw_width = fuji_width = fuji_layout = cr2_slice[0] = 0;
8632.  maximum = height = width = top_margin = left_margin = 0;
8633.  cdesc[0] = desc[0] = artist[0] = make[0] = model[0] = model2[0] = 0;
8634.  iso_speed = shutter = aperture = focal_len = unique_id = 0;
8635.  tiff_nifds = 0;
8636.  memset (tiff_ifd, 0, sizeof tiff_ifd);
8637.  memset (gpsdata, 0, sizeof gpsdata);
8638.  memset (cblack, 0, sizeof cblack);
```

```
8639.  memset (white, 0, sizeof white);
8640.  memset (mask, 0, sizeof mask);
8641.  thumb_offset = thumb_length = thumb_width = thumb_height = 0;
8642.  load_raw = thumb_load_raw = 0;
8643.  write_thumb = &CLASS jpeg_thumb;
8644.  data_offset = meta_offset = meta_length = tiff_bps = tiff_compress = 0;
8645.  kodak_cbpp = zero_after_ff = dng_version = load_flags = 0;
8646.  timestamp = shot_order = tiff_samples = black = is_foveon = 0;
8647.  mix_green = profile_length = data_error = zero_is_bad = 0;
8648.  pixel_aspect = is_raw = raw_color = 1;
8649.  tile_width = tile_length = 0;
8650.  for (i=0; i < 4; i++) {
8651.    cam_mul[i] = i == 1;
8652.    pre_mul[i] = i < 3;
8653.    FORC3 cmatrix[c][i] = 0;
8654.    FORC3 rgb_cam[c][i] = c == i;
8655.  }
8656.  colors = 3;
8657.  for (i=0; i < 0x10000; i++) curve[i] = i;
8658.
8659.  order = get2();
8660.  hlen = get4();
8661.  fseek (ifp, 0, SEEK_SET);
8662.  fread (head, 1, 32, ifp);
8663.  fseek (ifp, 0, SEEK_END);
8664.  flen = fsize = ftell(ifp);
8665.  if ((cp = (char *) memmem (head, 32, "MMMM", 4)) ||
8666.      (cp = (char *) memmem (head, 32, "IIII", 4))) {
8667.    parse_phase_one (cp-head);
8668.    if (cp-head && parse_tiff(0)) apply_tiff();
8669.  } else if (order == 0x4949 || order == 0x4d4d) {
8670.    if (!memcmp (head+6,"HEAPCCDR",8)) {
8671.      data_offset = hlen;
8672.      parse_ciff (hlen, flen-hlen, 0);
8673.      load_raw = &CLASS canon_load_raw;
8674.    } else if (parse_tiff(0)) apply_tiff();
8675.  } else if (!memcmp (head,"\xff\xd8\xff\xe1",4) &&
8676.             !memcmp (head+6,"Exif",4)) {
8677.    fseek (ifp, 4, SEEK_SET);
8678.    data_offset = 4 + get2();
8679.    fseek (ifp, data_offset, SEEK_SET);
8680.    if (fgetc(ifp) != 0xff)
8681.      parse_tiff(12);
8682.    thumb_offset = 0;
8683.  } else if (!memcmp (head+25,"ARECOYK",7)) {
8684.    strcpy (make, "Contax");
8685.    strcpy (model,"N Digital");
8686.    fseek (ifp, 33, SEEK_SET);
8687.    get_timestamp(1);
8688.    fseek (ifp, 60, SEEK_SET);
8689.    FORC4 cam_mul[c ^ (c >> 1)] = get4();
8690.  } else if (!strcmp (head, "PXN")) {
8691.    strcpy (make, "Logitech");
8692.    strcpy (model,"Fotoman Pixtura");
8693.  } else if (!strcmp (head, "qktk")) {
8694.    strcpy (make, "Apple");
8695.    strcpy (model,"QuickTake 100");
8696.    load_raw = &CLASS quicktake_100_load_raw;
8697.  } else if (!strcmp (head, "qktn")) {
8698.    strcpy (make, "Apple");
8699.    strcpy (model,"QuickTake 150");
8700.    load_raw = &CLASS kodak_radc_load_raw;
8701.  } else if (!memcmp (head,"FUJIFILM",8)) {
8702.    fseek (ifp, 84, SEEK_SET);
8703.    thumb_offset = get4();
```

```
8704.    thumb_length = get4();
8705.    fseek (ifp, 92, SEEK_SET);
8706.    parse_fuji (get4());
8707.    if (thumb_offset > 120) {
8708.      fseek (ifp, 120, SEEK_SET);
8709.      is_raw += (i = get4()) && 1;
8710.      if (is_raw == 2 && shot_select)
8711.        parse_fuji (i);
8712.    }
8713.    fseek (ifp, 100+28*(shot_select > 0), SEEK_SET);
8714.    parse_tiff (data_offset = get4());
8715.    parse_tiff (thumb_offset+12);
8716.    apply_tiff();
8717.    if (!load_raw) {
8718.      load_raw = &CLASS unpacked_load_raw;
8719.      tiff_bps = 14;
8720.    }
8721.  } else if (!memcmp (head,"RIFF",4)) {
8722.    fseek (ifp, 0, SEEK_SET);
8723.    parse_riff();
8724.  } else if (!memcmp (head+4,"ftypcrx ",8)) {
8725.    fseek (ifp, 0, SEEK_SET);
8726.    parse_crx (fsize);
8727.  } else if (!memcmp (head+4,"ftypqt   ",9)) {
8728.    fseek (ifp, 0, SEEK_SET);
8729.    parse_qt (fsize);
8730.    is_raw = 0;
8731.  } else if (!memcmp (head,"\0\001\0\001\0@",6)) {
8732.    fseek (ifp, 6, SEEK_SET);
8733.    fread (make, 1, 8, ifp);
8734.    fread (model, 1, 8, ifp);
8735.    fread (model2, 1, 16, ifp);
8736.    data_offset = get2();
8737.    get2();
8738.    raw_width = get2();
8739.    raw_height = get2();
8740.    load_raw = &CLASS nokia_load_raw;
8741.    filters = 0x61616161;
8742.  } else if (!memcmp (head,"NOKIARAW",8)) {
8743.    strcpy (make, "NOKIA");
8744.    order = 0x4949;
8745.    fseek (ifp, 300, SEEK_SET);
8746.    data_offset = get4();
8747.    i = get4();
8748.    width = get2();
8749.    height = get2();
8750.    switch (tiff_bps = i*8 / (width * height)) {
8751.      case  8: load_raw = &CLASS eight_bit_load_raw;  break;
8752.      case 10: load_raw = &CLASS nokia_load_raw;
8753.    }
8754.    raw_height = height + (top_margin = i / (width * tiff_bps/8) - height);
8755.    mask[0][3] = 1;
8756.    filters = 0x61616161;
8757.  } else if (!memcmp (head,"ARRI",4)) {
8758.    order = 0x4949;
8759.    fseek (ifp, 20, SEEK_SET);
8760.    width = get4();
8761.    height = get4();
8762.    strcpy (make, "ARRI");
8763.    fseek (ifp, 668, SEEK_SET);
8764.    fread (model, 1, 64, ifp);
8765.    data_offset = 4096;
8766.    load_raw = &CLASS packed_load_raw;
8767.    load_flags = 88;
8768.    filters = 0x61616161;
```

```
8769.  } else if (!memcmp (head,"XPDS",4)) {
8770.    order = 0x4949;
8771.    fseek (ifp, 0x800, SEEK_SET);
8772.    fread (make, 1, 41, ifp);
8773.    raw_height = get2();
8774.    raw_width  = get2();
8775.    fseek (ifp, 56, SEEK_CUR);
8776.    fread (model, 1, 30, ifp);
8777.    data_offset = 0x10000;
8778.    load_raw = &CLASS canon_rmf_load_raw;
8779.    gamma_curve (0, 12.25, 1, 1023);
8780.  } else if (!memcmp (head+4,"RED1",4)) {
8781.    strcpy (make, "Red");
8782.    strcpy (model,"One");
8783.    parse_redcine();
8784.    load_raw = &CLASS redcine_load_raw;
8785.    gamma_curve (1/2.4, 12.92, 1, 4095);
8786.    filters = 0x49494949;
8787.  } else if (!memcmp (head,"DSC-Image",9))
8788.    parse_rollei();
8789.  else if (!memcmp (head,"PWAD",4))
8790.    parse_sinar_ia();
8791.  else if (!memcmp (head,"\0MRM",4))
8792.    parse_minolta(0);
8793.  else if (!memcmp (head,"FOVb",4))
8794.    parse_foveon();
8795.  else if (!memcmp (head,"CI",2))
8796.    parse_cine();
8797.  if (make[0] == 0)
8798.    for (zero_fsize=i=0; i < sizeof table / sizeof *table; i++)
8799.      if (fsize == table[i].fsize) {
8800.        strcpy (make,  table[i].make );
8801.        strcpy (model, table[i].model);
8802.        flip = table[i].flags >> 2;
8803.        zero_is_bad = table[i].flags & 2;
8804.        if (table[i].flags & 1)
8805.          parse_external_jpeg();
8806.        data_offset = table[i].offset;
8807.        raw_width   = table[i].rw;
8808.        raw_height  = table[i].rh;
8809.        left_margin = table[i].lm;
8810.         top_margin = table[i].tm;
8811.        width  = raw_width - left_margin - table[i].rm;
8812.        height = raw_height - top_margin - table[i].bm;
8813.        filters = 0x1010101 * table[i].cf;
8814.        colors = 4 - !((filters & filters >> 1) & 0x5555);
8815.        load_flags = table[i].lf;
8816.        switch (tiff_bps = (fsize-data_offset)*8 / (raw_width*raw_height)) {
8817.          case 6:
8818.            load_raw = &CLASS minolta_rd175_load_raw;  break;
8819.          case 8:
8820.            load_raw = &CLASS eight_bit_load_raw;  break;
8821.          case 10: case 12:
8822.            load_flags |= 512;
8823.            if (!strcmp(make,"Canon")) load_flags |= 256;
8824.            load_raw = &CLASS packed_load_raw;     break;
8825.          case 16:
8826.            order = 0x4949 | 0x404 * (load_flags & 1);
8827.            tiff_bps -= load_flags >> 4;
8828.            tiff_bps -= load_flags = load_flags >> 1 & 7;
8829.            load_raw = &CLASS unpacked_load_raw;
8830.        }
8831.        maximum = (1 << tiff_bps) - (1 << table[i].max);
8832.      }
8833.  if (zero_fsize) fsize = 0;
```

```
8834.  if (make[0] == 0) parse_smal (0, flen);
8835.  if (make[0] == 0) {
8836.    parse_jpeg(0);
8837.    if (!(strncmp(model,"ov",2) && strncmp(model,"RP_OV",5)) &&
8838.        !fseek (ifp, -6404096, SEEK_END) &&
8839.        fread (head, 1, 32, ifp) && !strcmp(head,"BRCMn")) {
8840.      strcpy (make, "OmniVision");
8841.      data_offset = ftell(ifp) + 0x8000-32;
8842.      width = raw_width;
8843.      raw_width = 2611;
8844.      load_raw = &CLASS nokia_load_raw;
8845.      filters = 0x16161616;
8846.    } else is_raw = 0;
8847.  }
8848.
8849.  for (i=0; i < sizeof corp / sizeof *corp; i++)
8850.    if (strcasestr (make, corp[i]))     /* Simplify company names */
8851.            strcpy (make, corp[i]);
8852.  if ((!strcmp(make,"Kodak") || !strcmp(make,"Leica")) &&
8853.       ((cp = strcasestr(model," DIGITAL CAMERA")) ||
8854.        (cp = strstr(model,"FILE VERSION"))))
8855.     *cp = 0;
8856.  if (!strncasecmp(model,"PENTAX",6))
8857.    strcpy (make, "Pentax");
8858.  cp = make + strlen(make);             /* Remove trailing spaces */
8859.  while (*--cp == ' ') *cp = 0;
8860.  cp = model + strlen(model);
8861.  while (*--cp == ' ') *cp = 0;
8862.  i = strlen(make);                     /* Remove make from model */
8863.  if (!strncasecmp (model, make, i) && model[i++] == ' ')
8864.    memmove (model, model+i, 64-i);
8865.  if (!strncmp (model,"FinePix ",8))
8866.    strcpy (model, model+8);
8867.  if (!strncmp (model,"Digital Camera ",15))
8868.    strcpy (model, model+15);
8869.  desc[511] = artist[63] = make[63] = model[63] = model2[63] = 0;
8870.  if (!is_raw) goto notraw;
8871.
8872.  if (!height) height = raw_height;
8873.  if (!width)  width  = raw_width;
8874.  if (height == 2624 && width == 3936)  /* Pentax K10D and Samsung GX10 */
8875.    { height  = 2616;   width  = 3896; }
8876.  if (height == 3136 && width == 4864)  /* Pentax K20D and Samsung GX20 */
8877.    { height  = 3124;   width  = 4688; filters = 0x16161616; }
8878.  if (raw_height == 2868 && (!strcmp(model,"K-r") || !strcmp(model,"K-x")))
8879.    {                   width  = 4309; filters = 0x16161616; }
8880.  if (raw_height == 3136 && !strcmp(model,"K-7"))
8881.    { height  = 3122;   width  = 4684; filters = 0x16161616; top_margin = 2; }
8882.  if (raw_height == 3284 && !strncmp(model,"K-5",3))
8883.    { left_margin = 10; width  = 4950; filters = 0x16161616; }
8884.  if (raw_height == 3300 && !strncmp(model,"K-50",4))
8885.    { height  = 3288,   width  = 4952;  left_margin = 0;  top_margin = 12; }
8886.  if (raw_height == 3664 && !strncmp(model,"K-S",3))
8887.    {                   width  = 5492;  left_margin = 0; }
8888.  if (raw_height == 4032 && !strcmp(model,"K-3"))
8889.    { height  = 4032;   width  = 6040;  left_margin = 4; }
8890.  if (raw_height == 4060 && !strcmp(model,"KP"))
8891.    { height  = 4032;   width  = 6032;  left_margin = 52; top_margin = 28; }
8892.  if (raw_height == 4950 && !strcmp(model,"K-1"))
8893.    { height  = 4932;   width  = 7380;  left_margin = 4;  top_margin = 18; }
8894.  if (raw_height == 5552 && !strcmp(model,"645D"))
8895.    { height  = 5502;   width  = 7328;  left_margin = 48; top_margin = 29;
8896.      filters = 0x61616161; }
8897.  if (height == 3014 && width == 4096)  /* Ricoh GX200 */
8898.                        width  = 4014;
```

```
8899.  if (dng_version) {
8900.    if (filters == UINT_MAX) filters = 0;
8901.    if (filters) is_raw *= tiff_samples;
8902.    else         colors  = tiff_samples;
8903.    switch (tiff_compress) {
8904.      case 0:
8905.      case 1:     load_raw = &CLASS   packed_dng_load_raw;  break;
8906.      case 7:     load_raw = &CLASS lossless_dng_load_raw;  break;
8907.      case 34892: load_raw = &CLASS    lossy_dng_load_raw;  break;
8908.      default:    load_raw = 0;
8909.    }
8910.    goto dng_skip;
8911.  }
8912.  if (!strcmp(make,"Canon") && !fsize && tiff_bps != 15) {
8913.    if (!load_raw)
8914.      load_raw = &CLASS lossless_jpeg_load_raw;
8915.    for (i=0; i < sizeof canon / sizeof *canon; i++)
8916.      if (raw_width == canon[i][0] && raw_height == canon[i][1]) {
8917.        width  = raw_width - (left_margin = canon[i][2]);
8918.        height = raw_height - (top_margin = canon[i][3]);
8919.        width  -= canon[i][4];
8920.        height -= canon[i][5];
8921.        mask[0][1] =  canon[i][6];
8922.        mask[0][3] = -canon[i][7];
8923.        mask[1][1] =  canon[i][8];
8924.        mask[1][3] = -canon[i][9];
8925.        if (canon[i][10]) filters = canon[i][10] * 0x01010101;
8926.      }
8927.    if ((unique_id | 0x20000) == 0x2720000) {
8928.      left_margin = 8;
8929.      top_margin = 16;
8930.    }
8931.  }
8932.  for (i=0; i < sizeof unique / sizeof *unique; i++)
8933.    if (unique_id == 0x80000000 + unique[i].id) {
8934.      adobe_coeff ("Canon", unique[i].model);
8935.      if (model[4] == 'K' && strlen(model) == 8)
8936.        strcpy (model, unique[i].model);
8937.    }
8938.  for (i=0; i < sizeof sonique / sizeof *sonique; i++)
8939.    if (unique_id == sonique[i].id)
8940.      strcpy (model, sonique[i].model);
8941.  for (i=0; i < sizeof panalias / sizeof *panalias; i++)
8942.    if (panalias[i][0] == '@') orig = panalias[i]+1;
8943.    else if (!strcmp(model,panalias[i]))
8944.      adobe_coeff ("Panasonic", orig);
8945.  if (!strcmp(make,"Nikon")) {
8946.    if (!load_raw)
8947.      load_raw = &CLASS packed_load_raw;
8948.    if (model[0] == 'E')
8949.      load_flags |= !data_offset << 2 | 2;
8950.  }
8951.
8952./* Set parameters based on camera name (for non-DNG files). */
8953.
8954.  if (!strcmp(model,"KAI-0340")
8955.        && find_green (16, 16, 3840, 5120) < 25) {
8956.    height = 480;
8957.    top_margin = filters = 0;
8958.    strcpy (model,"C603");
8959.  }
8960.  if (!strcmp(make,"Sony") && raw_width > 3888)
8961.    black = 128 << (tiff_bps - 12);
8962.  if (is_foveon) {
8963.    if (height*2 < width) pixel_aspect = 0.5;
```

```
8964.     if (height   > width) pixel_aspect = 2;
8965.     filters = 0;
8966.     simple_coeff(0);
8967.   } else if (!strcmp(make,"Canon") && tiff_bps == 15) {
8968.     switch (width) {
8969.       case 3344: width -= 66;
8970.       case 3872: width -= 6;
8971.     }
8972.     if (height > width) {
8973.       SWAP(height,width);
8974.       SWAP(raw_height,raw_width);
8975.     }
8976.     if (width == 7200 && height == 3888) {
8977.       raw_width  = width  = 6480;
8978.       raw_height = height = 4320;
8979.     }
8980.     filters = 0;
8981.     tiff_samples = colors = 3;
8982.     load_raw = &CLASS canon_sraw_load_raw;
8983.   } else if (!strcmp(model,"PowerShot 600")) {
8984.     height = 613;
8985.     width  = 854;
8986.     raw_width = 896;
8987.     colors = 4;
8988.     filters = 0xe1e4e1e4;
8989.     load_raw = &CLASS canon_600_load_raw;
8990.   } else if (!strcmp(model,"PowerShot A5") ||
8991.              !strcmp(model,"PowerShot A5 Zoom")) {
8992.     height = 773;
8993.     width  = 960;
8994.     raw_width = 992;
8995.     pixel_aspect = 256/235.0;
8996.     filters = 0x1e4e1e4e;
8997.     goto canon_a5;
8998.   } else if (!strcmp(model,"PowerShot A50")) {
8999.     height =  968;
9000.     width  = 1290;
9001.     raw_width = 1320;
9002.     filters = 0x1b4e4b1e;
9003.     goto canon_a5;
9004.   } else if (!strcmp(model,"PowerShot Pro70")) {
9005.     height = 1024;
9006.     width  = 1552;
9007.     filters = 0x1e4b4e1b;
9008. canon_a5:
9009.     colors = 4;
9010.     tiff_bps = 10;
9011.     load_raw = &CLASS packed_load_raw;
9012.     load_flags = 264;
9013.   } else if (!strcmp(model,"PowerShot Pro90 IS") ||
9014.              !strcmp(model,"PowerShot G1")) {
9015.     colors = 4;
9016.     filters = 0xb4b4b4b4;
9017.   } else if (!strcmp(model,"PowerShot A610")) {
9018.     if (canon_s2is()) strcpy (model+10, "S2 IS");
9019.   } else if (!strcmp(model,"PowerShot SX220 HS")) {
9020.     mask[1][3] = -4;
9021.   } else if (!strcmp(model,"EOS D2000C")) {
9022.     filters = 0x61616161;
9023.     black = curve[200];
9024.   } else if (!strcmp(model,"EOS 80D")) {
9025.     top_margin -= 2;
9026.     height += 2;
9027.   } else if (!strcmp(model,"D1")) {
9028.     cam_mul[0] *= 256/527.0;
```

```
    cam_mul[2] *= 256/317.0;
  } else if (!strcmp(model,"D1X")) {
    width -= 4;
    pixel_aspect = 0.5;
  } else if (!strcmp(model,"D40X") ||
             !strcmp(model,"D60")  ||
             !strcmp(model,"D80")  ||
             !strcmp(model,"D3000")) {
    height -= 3;
    width  -= 4;
  } else if (!strcmp(model,"D3")   ||
             !strcmp(model,"D3S")  ||
             !strcmp(model,"D700")) {
    width -= 4;
    left_margin = 2;
  } else if (!strcmp(model,"D3100")) {
    width -= 28;
    left_margin = 6;
  } else if (!strcmp(model,"D5000") ||
             !strcmp(model,"D90")) {
    width -= 42;
  } else if (!strcmp(model,"D5100") ||
             !strcmp(model,"D7000") ||
             !strcmp(model,"COOLPIX A")) {
    width -= 44;
  } else if (!strcmp(model,"D3200") ||
            !strncmp(model,"D6",2)  ||
            !strncmp(model,"D800",4)) {
    width -= 46;
  } else if (!strcmp(model,"D4") ||
             !strcmp(model,"Df")) {
    width -= 52;
    left_margin = 2;
  } else if (!strncmp(model,"D40",3) ||
             !strncmp(model,"D50",3) ||
             !strncmp(model,"D70",3)) {
    width--;
  } else if (!strcmp(model,"D100")) {
    if (load_flags)
      raw_width = (width += 3) + 3;
  } else if (!strcmp(model,"D200")) {
    left_margin = 1;
    width -= 4;
    filters = 0x94949494;
  } else if (!strncmp(model,"D2H",3)) {
    left_margin = 6;
    width -= 14;
  } else if (!strncmp(model,"D2X",3)) {
    if (width == 3264) width -= 32;
    else width -= 8;
  } else if (!strncmp(model,"D300",4)) {
    width -= 32;
  } else if (!strncmp(model,"COOLPIX B",9)) {
    load_flags = 24;
  } else if (!strncmp(model,"COOLPIX P",9) && raw_width != 4032) {
    load_flags = 24;
    filters = 0x94949494;
    if (model[9] == '7' && iso_speed >= 400)
      black = 255;
  } else if (!strncmp(model,"1 ",2)) {
    height -= 2;
  } else if (fsize == 1581060) {
    simple_coeff(3);
    pre_mul[0] = 1.2085;
    pre_mul[1] = 1.0943;
```

```
    pre_mul[3] = 1.1103;
  } else if (fsize == 3178560) {
    cam_mul[0] *= 4;
    cam_mul[2] *= 4;
  } else if (fsize == 4771840) {
    if (!timestamp && nikon_e995())
      strcpy (model, "E995");
    if (strcmp(model,"E995")) {
      filters = 0xb4b4b4b4;
      simple_coeff(3);
      pre_mul[0] = 1.196;
      pre_mul[1] = 1.246;
      pre_mul[2] = 1.018;
    }
  } else if (fsize == 2940928) {
    if (!timestamp && !nikon_e2100())
      strcpy (model,"E2500");
    if (!strcmp(model,"E2500")) {
      height -= 2;
      load_flags = 6;
      colors = 4;
      filters = 0x4b4b4b4b;
    }
  } else if (fsize == 4775936) {
    if (!timestamp) nikon_3700();
    if (model[0] == 'E' && atoi(model+1) < 3700)
      filters = 0x49494949;
    if (!strcmp(model,"Optio 33WR")) {
      flip = 1;
      filters = 0x16161616;
    }
    if (make[0] == 'O') {
      i = find_green (12, 32, 1188864, 3576832);
      c = find_green (12, 32, 2383920, 2387016);
      if (abs(i) < abs(c)) {
        SWAP(i,c);
        load_flags = 24;
      }
      if (i < 0) filters = 0x61616161;
    }
  } else if (fsize == 5869568) {
    if (!timestamp && minolta_z2()) {
      strcpy (make, "Minolta");
      strcpy (model,"DiMAGE Z2");
    }
    load_flags = 6 + 24*(make[0] == 'M');
  } else if (fsize == 6291456) {
    fseek (ifp, 0x300000, SEEK_SET);
    if ((order = guess_byte_order(0x10000)) == 0x4d4d) {
      height -= (top_margin = 16);
      width -= (left_margin = 28);
      maximum = 0xf5c0;
      strcpy (make, "ISG");
      model[0] = 0;
    }
  } else if (!strcmp(make,"Fujifilm")) {
    if (!strcmp(model+7,"S2Pro")) {
      strcpy (model,"S2Pro");
      height = 2144;
      width  = 2880;
      flip = 6;
    }
    top_margin = (raw_height - height) >> 2 << 1;
    left_margin = (raw_width - width ) >> 2 << 1;
    if (width == 2848 || width == 3664) filters = 0x16161616;
```

```
9159.     if (width == 4032 || width == 4952 || width == 6032 || width == 8280) left_margin =
      0;
9160.     if (width == 3328 && (width -= 66)) left_margin = 34;
9161.     if (width == 4936) left_margin = 4;
9162.     if (!strcmp(model,"HS50EXR") ||
9163.         !strcmp(model,"F900EXR")) {
9164.       width += 2;
9165.       left_margin = 0;
9166.       filters = 0x16161616;
9167.     }
9168.     if (fuji_layout) raw_width *= is_raw;
9169.     if (filters == 9)
9170.       FORC(36) ((char *)xtrans)[c] =
9171.         xtrans_abs[(c/6+top_margin) % 6][(c+left_margin) % 6];
9172.   } else if (!strcmp(model,"KD-400Z")) {
9173.     height = 1712;
9174.     width  = 2312;
9175.     raw_width = 2336;
9176.     goto konica_400z;
9177.   } else if (!strcmp(model,"KD-510Z")) {
9178.     goto konica_510z;
9179.   } else if (!strcasecmp(make,"Minolta")) {
9180.     if (!load_raw && (maximum = 0xfff))
9181.       load_raw = &CLASS unpacked_load_raw;
9182.     if (!strncmp(model,"DiMAGE A",8)) {
9183.       if (!strcmp(model,"DiMAGE A200"))
9184.         filters = 0x49494949;
9185.       tiff_bps = 12;
9186.       load_raw = &CLASS packed_load_raw;
9187.     } else if (!strncmp(model,"ALPHA",5) ||
9188.                !strncmp(model,"DYNAX",5) ||
9189.                !strncmp(model,"MAXXUM",6)) {
9190.       sprintf (model+20, "DYNAX %-10s", model+6+(model[0]=='M'));
9191.       adobe_coeff (make, model+20);
9192.       load_raw = &CLASS packed_load_raw;
9193.     } else if (!strncmp(model,"DiMAGE G",8)) {
9194.       if (model[8] == '4') {
9195.         height = 1716;
9196.         width  = 2304;
9197.       } else if (model[8] == '5') {
9198. konica_510z:
9199.         height = 1956;
9200.         width  = 2607;
9201.         raw_width = 2624;
9202.       } else if (model[8] == '6') {
9203.         height = 2136;
9204.         width  = 2848;
9205.       }
9206.       data_offset += 14;
9207.       filters = 0x61616161;
9208. konica_400z:
9209.       load_raw = &CLASS unpacked_load_raw;
9210.       maximum = 0x3df;
9211.       order = 0x4d4d;
9212.     }
9213.   } else if (!strcmp(model,"*ist D")) {
9214.     load_raw = &CLASS unpacked_load_raw;
9215.     data_error = -1;
9216.   } else if (!strcmp(model,"*ist DS")) {
9217.     height -= 2;
9218.   } else if (!strcmp(make,"Samsung") && raw_width == 4704) {
9219.     height -= top_margin = 8;
9220.     width -= 2 * (left_margin = 8);
9221.     load_flags = 256;
9222.   } else if (!strcmp(make,"Samsung") && raw_height == 3714) {
```

```
9223.    height -= top_margin = 18;
9224.    left_margin = raw_width - (width = 5536);
9225.    if (raw_width != 5600)
9226.      left_margin = top_margin = 0;
9227.    filters = 0x61616161;
9228.    colors = 3;
9229.  } else if (!strcmp(make,"Samsung") && raw_width == 5632) {
9230.    order = 0x4949;
9231.    height = 3694;
9232.    top_margin = 2;
9233.    width  = 5574 - (left_margin = 32 + tiff_bps);
9234.    if (tiff_bps == 12) load_flags = 80;
9235.  } else if (!strcmp(make,"Samsung") && raw_width == 5664) {
9236.    height -= top_margin = 17;
9237.    left_margin = 96;
9238.    width = 5544;
9239.    filters = 0x49494949;
9240.  } else if (!strcmp(make,"Samsung") && raw_width == 6496) {
9241.    filters = 0x61616161;
9242.    black = 1 << (tiff_bps - 7);
9243.  } else if (!strcmp(model,"EX1")) {
9244.    order = 0x4949;
9245.    height -= 20;
9246.    top_margin = 2;
9247.    if ((width -= 6) > 3682) {
9248.      height -= 10;
9249.      width  -= 46;
9250.      top_margin = 8;
9251.    }
9252.  } else if (!strcmp(model,"WB2000")) {
9253.    order = 0x4949;
9254.    height -= 3;
9255.    top_margin = 2;
9256.    if ((width -= 10) > 3718) {
9257.      height -= 28;
9258.      width  -= 56;
9259.      top_margin = 8;
9260.    }
9261.  } else if (strstr(model,"WB550")) {
9262.    strcpy (model, "WB550");
9263.  } else if (!strcmp(model,"EX2F")) {
9264.    height = 3045;
9265.    width  = 4070;
9266.    top_margin = 3;
9267.    order = 0x4949;
9268.    filters = 0x49494949;
9269.    load_raw = &CLASS unpacked_load_raw;
9270.  } else if (!strcmp(model,"STV680 VGA")) {
9271.    black = 16;
9272.  } else if (!strcmp(model,"N95")) {
9273.    height = raw_height - (top_margin = 2);
9274.  } else if (!strcmp(model,"640x480")) {
9275.    gamma_curve (0.45, 4.5, 1, 255);
9276.  } else if (!strcmp(make,"Hasselblad")) {
9277.    if (load_raw == &CLASS lossless_jpeg_load_raw)
9278.      load_raw = &CLASS hasselblad_load_raw;
9279.    if (raw_width == 7262) {
9280.      height = 5444;
9281.      width  = 7248;
9282.      top_margin  = 4;
9283.      left_margin = 7;
9284.      filters = 0x61616161;
9285.    } else if (raw_width == 7410 || raw_width == 8282) {
9286.      height -= 84;
9287.      width  -= 82;
```

```
9288.       top_margin  = 4;
9289.       left_margin = 41;
9290.       filters = 0x61616161;
9291.     } else if (raw_width == 8384) {
9292.       height = 6208;
9293.       width  = 8280;
9294.       top_margin  = 96;
9295.       left_margin = 46;
9296.     } else if (raw_width == 9044) {
9297.       height = 6716;
9298.       width  = 8964;
9299.       top_margin  = 8;
9300.       left_margin = 40;
9301.       black += load_flags = 256;
9302.       maximum = 0x8101;
9303.     } else if (raw_width == 4090) {
9304.       strcpy (model, "V96C");
9305.       height -= (top_margin = 6);
9306.       width -= (left_margin = 3) + 7;
9307.       filters = 0x61616161;
9308.     }
9309.     if (tiff_samples > 1) {
9310.       is_raw = tiff_samples+1;
9311.       if (!shot_select && !half_size) filters = 0;
9312.     }
9313.   } else if (!strcmp(make,"Sinar")) {
9314.     if (!load_raw) load_raw = &CLASS unpacked_load_raw;
9315.     if (is_raw > 1 && !shot_select && !half_size) filters = 0;
9316.     maximum = 0x3fff;
9317.   } else if (!strcmp(make,"Leaf")) {
9318.     maximum = 0x3fff;
9319.     fseek (ifp, data_offset, SEEK_SET);
9320.     if (ljpeg_start (&jh, 1) && jh.bits == 15)
9321.       maximum = 0x1fff;
9322.     if (tiff_samples > 1) filters = 0;
9323.     if (tiff_samples > 1 || tile_length < raw_height) {
9324.       load_raw = &CLASS leaf_hdr_load_raw;
9325.       raw_width = tile_width;
9326.     }
9327.     if ((width | height) == 2048) {
9328.       if (tiff_samples == 1) {
9329.         filters = 1;
9330.         strcpy (cdesc, "RBTG");
9331.         strcpy (model, "CatchLight");
9332.         top_margin =  8; left_margin = 18; height = 2032; width = 2016;
9333.       } else {
9334.         strcpy (model, "DCB2");
9335.         top_margin = 10; left_margin = 16; height = 2028; width = 2022;
9336.       }
9337.     } else if (width+height == 3144+2060) {
9338.       if (!model[0]) strcpy (model, "Cantare");
9339.       if (width > height) {
9340.         top_margin = 6; left_margin = 32; height = 2048;  width = 3072;
9341.         filters = 0x61616161;
9342.       } else {
9343.         left_margin = 6;  top_margin = 32;  width = 2048; height = 3072;
9344.         filters = 0x16161616;
9345.       }
9346.       if (!cam_mul[0] || model[0] == 'V') filters = 0;
9347.       else is_raw = tiff_samples;
9348.     } else if (width == 2116) {
9349.       strcpy (model, "Valeo 6");
9350.       height -= 2 * (top_margin = 30);
9351.       width -= 2 * (left_margin = 55);
9352.       filters = 0x49494949;
```

```
9353.    } else if (width == 3171) {
9354.      strcpy (model, "Valeo 6");
9355.      height -= 2 * (top_margin = 24);
9356.      width -= 2 * (left_margin = 24);
9357.      filters = 0x16161616;
9358.    }
9359.  } else if (!strcmp(make,"Leica") || !strcmp(make,"Panasonic")) {
9360.    if ((flen - data_offset) / (raw_width*8/7) == raw_height)
9361.      load_raw = &CLASS panasonic_load_raw;
9362.    if (!load_raw) {
9363.      load_raw = &CLASS unpacked_load_raw;
9364.      load_flags = 4;
9365.    }
9366.    zero_is_bad = 1;
9367.    if ((height += 12) > raw_height) height = raw_height;
9368.    for (i=0; i < sizeof pana / sizeof *pana; i++)
9369.      if (raw_width == pana[i][0] && raw_height == pana[i][1]) {
9370.        left_margin = pana[i][2];
9371.         top_margin = pana[i][3];
9372.            width += pana[i][4];
9373.           height += pana[i][5];
9374.      }
9375.    filters = 0x01010101 * (uchar) "\x94\x61\x49\x16"
9376.        [((filters-1) ^ (left_margin & 1) ^ (top_margin << 1)) & 3];
9377.  } else if (!strcmp(model,"C770UZ")) {
9378.    height = 1718;
9379.    width  = 2304;
9380.    filters = 0x16161616;
9381.    load_raw = &CLASS packed_load_raw;
9382.    load_flags = 30;
9383.  } else if (!strcmp(make,"Olympus")) {
9384.    height += height & 1;
9385.    if (exif_cfa) filters = exif_cfa;
9386.    if (width == 4100) width -= 4;
9387.    if (width == 4080) width -= 24;
9388.    if (width == 9280) { width -= 6; height -= 6; }
9389.    if (load_raw == &CLASS unpacked_load_raw)
9390.      load_flags = 4;
9391.    tiff_bps = 12;
9392.    if (!strcmp(model,"E-300") ||
9393.        !strcmp(model,"E-500")) {
9394.      width -= 20;
9395.      if (load_raw == &CLASS unpacked_load_raw) {
9396.        maximum = 0xfc3;
9397.        memset (cblack, 0, sizeof cblack);
9398.      }
9399.    } else if (!strcmp(model,"E-330")) {
9400.      width -= 30;
9401.      if (load_raw == &CLASS unpacked_load_raw)
9402.        maximum = 0xf79;
9403.    } else if (!strcmp(model,"SP550UZ")) {
9404.      thumb_length = flen - (thumb_offset = 0xa39800);
9405.      thumb_height = 480;
9406.      thumb_width  = 640;
9407.    } else if (!strcmp(model,"TG-4")) {
9408.      width -= 16;
9409.    } else if (!strcmp(model,"TG-5")) {
9410.      width -= 6;
9411.    }
9412.  } else if (!strcmp(model,"N Digital")) {
9413.    height = 2047;
9414.    width  = 3072;
9415.    filters = 0x61616161;
9416.    data_offset = 0x1a00;
9417.    load_raw = &CLASS packed_load_raw;
```

```
9418.  } else if (!strcmp(model,"DSC-F828")) {
9419.    width = 3288;
9420.    left_margin = 5;
9421.    mask[1][3] = -17;
9422.    data_offset = 862144;
9423.    load_raw = &CLASS sony_load_raw;
9424.    filters = 0x9c9c9c9c;
9425.    colors = 4;
9426.    strcpy (cdesc, "RGBE");
9427.  } else if (!strcmp(model,"DSC-V3")) {
9428.    width = 3109;
9429.    left_margin = 59;
9430.    mask[0][1] = 9;
9431.    data_offset = 787392;
9432.    load_raw = &CLASS sony_load_raw;
9433.  } else if (!strcmp(make,"Sony") && raw_width == 3984) {
9434.    width = 3925;
9435.    order = 0x4d4d;
9436.  } else if (!strcmp(make,"Sony") && raw_width == 4288) {
9437.    width -= 32;
9438.  } else if (!strcmp(make,"Sony") && raw_width == 4600) {
9439.    if (!strcmp(model,"DSLR-A350"))
9440.      height -= 4;
9441.    black = 0;
9442.  } else if (!strcmp(make,"Sony") && raw_width == 4928) {
9443.    if (height < 3280) width -= 8;
9444.  } else if (!strcmp(make,"Sony") && raw_width == 5504) {
9445.    width -= height > 3664 ? 8 : 32;
9446.    if (!strncmp(model,"DSC",3))
9447.      black = 200 << (tiff_bps - 12);
9448.  } else if (!strcmp(make,"Sony") && raw_width == 6048) {
9449.    width -= 24;
9450.    if (strstr(model,"RX1") || strstr(model,"A99"))
9451.      width -= 6;
9452.  } else if (!strcmp(make,"Sony") && raw_width == 7392) {
9453.    width -= 30;
9454.  } else if (!strcmp(make,"Sony") && raw_width == 8000) {
9455.    width -= 32;
9456.  } else if (!strcmp(model,"DSLR-A100")) {
9457.    if (width == 3880) {
9458.      height--;
9459.      width = ++raw_width;
9460.    } else {
9461.      height -= 4;
9462.      width  -= 4;
9463.      order = 0x4d4d;
9464.      load_flags = 2;
9465.    }
9466.    filters = 0x61616161;
9467.  } else if (!strcmp(model,"PIXL")) {
9468.    height -= top_margin = 4;
9469.    width -= left_margin = 32;
9470.    gamma_curve (0, 7, 1, 255);
9471.  } else if (!strcmp(model,"C603") || !strcmp(model,"C330")
9472.      || !strcmp(model,"12MP")) {
9473.    order = 0x4949;
9474.    if (filters && data_offset) {
9475.      fseek (ifp, data_offset < 4096 ? 168 : 5252, SEEK_SET);
9476.      read_shorts (curve, 256);
9477.    } else gamma_curve (0, 3.875, 1, 255);
9478.    load_raw  =  filters   ? &CLASS eight_bit_load_raw :
9479.      strcmp(model,"C330") ? &CLASS kodak_c603_load_raw :
9480.                             &CLASS kodak_c330_load_raw;
9481.    load_flags = tiff_bps > 16;
9482.    tiff_bps = 8;
```

```
9483.  } else if (!strncasecmp(model,"EasyShare",9)) {
9484.    data_offset = data_offset < 0x15000 ? 0x15000 : 0x17000;
9485.    load_raw = &CLASS packed_load_raw;
9486.  } else if (!strcasecmp(make,"Kodak")) {
9487.    if (filters == UINT_MAX) filters = 0x61616161;
9488.    if (!strncmp(model,"NC2000",6) ||
9489.        !strncmp(model,"EOSDCS",6) ||
9490.        !strncmp(model,"DCS4",4)) {
9491.      width -= 4;
9492.      left_margin = 2;
9493.      if (model[6] == ' ') model[6] = 0;
9494.      if (!strcmp(model,"DCS460A")) goto bw;
9495.    } else if (!strcmp(model,"DCS660M")) {
9496.      black = 214;
9497.      goto bw;
9498.    } else if (!strcmp(model,"DCS760M")) {
9499.bw:   colors = 1;
9500.      filters = 0;
9501.    }
9502.    if (!strcmp(model+4,"20X"))
9503.      strcpy (cdesc, "MYCY");
9504.    if (strstr(model,"DC25")) {
9505.      strcpy (model, "DC25");
9506.      data_offset = 15424;
9507.    }
9508.    if (!strncmp(model,"DC2",3)) {
9509.      raw_height = 2 + (height = 242);
9510.      if (flen < 100000) {
9511.        raw_width = 256; width = 249;
9512.        pixel_aspect = (4.0*height) / (3.0*width);
9513.      } else {
9514.        raw_width = 512; width = 501;
9515.        pixel_aspect = (493.0*height) / (373.0*width);
9516.      }
9517.      top_margin = left_margin = 1;
9518.      colors = 4;
9519.      filters = 0x8d8d8d8d;
9520.      simple_coeff(1);
9521.      pre_mul[1] = 1.179;
9522.      pre_mul[2] = 1.209;
9523.      pre_mul[3] = 1.036;
9524.      load_raw = &CLASS eight_bit_load_raw;
9525.    } else if (!strcmp(model,"40")) {
9526.      strcpy (model, "DC40");
9527.      height = 512;
9528.      width  = 768;
9529.      data_offset = 1152;
9530.      load_raw = &CLASS kodak_radc_load_raw;
9531.      tiff_bps = 12;
9532.    } else if (strstr(model,"DC50")) {
9533.      strcpy (model, "DC50");
9534.      height = 512;
9535.      width  = 768;
9536.      data_offset = 19712;
9537.      load_raw = &CLASS kodak_radc_load_raw;
9538.    } else if (strstr(model,"DC120")) {
9539.      strcpy (model, "DC120");
9540.      height = 976;
9541.      width  = 848;
9542.      pixel_aspect = height/0.75/width;
9543.      load_raw = tiff_compress == 7 ?
9544.        &CLASS kodak_jpeg_load_raw : &CLASS kodak_dc120_load_raw;
9545.    } else if (!strcmp(model,"DCS200")) {
9546.      thumb_height = 128;
9547.      thumb_width  = 192;
```

```
9548.       thumb_offset = 6144;
9549.       thumb_misc   = 360;
9550.       write_thumb = &CLASS layer_thumb;
9551.       black = 17;
9552.     }
9553.   } else if (!strcmp(model,"Fotoman Pixtura")) {
9554.     height = 512;
9555.     width  = 768;
9556.     data_offset = 3632;
9557.     load_raw = &CLASS kodak_radc_load_raw;
9558.     filters = 0x61616161;
9559.     simple_coeff(2);
9560.   } else if (!strncmp(model,"QuickTake",9)) {
9561.     if (head[5]) strcpy (model+10, "200");
9562.     fseek (ifp, 544, SEEK_SET);
9563.     height = get2();
9564.     width  = get2();
9565.     data_offset = (get4(),get2()) == 30 ? 738:736;
9566.     if (height > width) {
9567.       SWAP(height,width);
9568.       fseek (ifp, data_offset-6, SEEK_SET);
9569.       flip = ~get2() & 3 ? 5:6;
9570.     }
9571.     filters = 0x61616161;
9572.   } else if (!strcmp(make,"Rollei") && !load_raw) {
9573.     switch (raw_width) {
9574.       case 1316:
9575.         height = 1030;
9576.         width  = 1300;
9577.         top_margin  = 1;
9578.         left_margin = 6;
9579.         break;
9580.       case 2568:
9581.         height = 1960;
9582.         width  = 2560;
9583.         top_margin  = 2;
9584.         left_margin = 8;
9585.     }
9586.     filters = 0x16161616;
9587.     load_raw = &CLASS rollei_load_raw;
9588.   }
9589.   if (!model[0])
9590.     sprintf (model, "%dx%d", width, height);
9591.   if (filters == UINT_MAX) filters = 0x94949494;
9592.   if (thumb_offset && !thumb_height) {
9593.     fseek (ifp, thumb_offset, SEEK_SET);
9594.     if (ljpeg_start (&jh, 1)) {
9595.       thumb_width  = jh.wide;
9596.       thumb_height = jh.high;
9597.     }
9598.   }
9599. dng_skip:
9600.   if ((use_camera_matrix & (use_camera_wb || dng_version))
9601.       && cmatrix[0][0] > 0.125) {
9602.     memcpy (rgb_cam, cmatrix, sizeof cmatrix);
9603.     raw_color = 0;
9604.   }
9605.   if (raw_color) adobe_coeff (make, model);
9606.   if (load_raw == &CLASS kodak_radc_load_raw)
9607.     if (raw_color) adobe_coeff ("Apple","Quicktake");
9608.   if (fuji_width) {
9609.     fuji_width = width >> !fuji_layout;
9610.     filters = fuji_width & 1 ? 0x94949494 : 0x49494949;
9611.     width = (height >> fuji_layout) + fuji_width;
9612.     height = width - 1;
```

```
9613.    pixel_aspect = 1;
9614.  } else {
9615.    if (raw_height < height) raw_height = height;
9616.    if (raw_width  < width ) raw_width  = width;
9617.  }
9618.  if (!tiff_bps) tiff_bps = 12;
9619.  if (!maximum) maximum = (1 << tiff_bps) - 1;
9620.  if (!load_raw || height < 22 || width < 22 ||
9621.        tiff_bps > 16 || tiff_samples > 6 || colors > 4)
9622.    is_raw = 0;
9623.#ifdef NO_JASPER
9624.  if (load_raw == &CLASS redcine_load_raw) {
9625.    fprintf (stderr,_("%s: You must link dcraw with %s!!\n"),
9626.        ifname, "libjasper");
9627.    is_raw = 0;
9628.  }
9629.#endif
9630.#ifdef NO_JPEG
9631.  if (load_raw == &CLASS kodak_jpeg_load_raw ||
9632.      load_raw == &CLASS lossy_dng_load_raw) {
9633.    fprintf (stderr,_("%s: You must link dcraw with %s!!\n"),
9634.        ifname, "libjpeg");
9635.    is_raw = 0;
9636.  }
9637.#endif
9638.  if (!cdesc[0])
9639.    strcpy (cdesc, colors == 3 ? "RGBG":"GMCY");
9640.  if (!raw_height) raw_height = height;
9641.  if (!raw_width ) raw_width  = width;
9642.  if (filters > 999 && colors == 3)
9643.    filters |= ((filters >> 2 & 0x22222222) |
9644.                (filters << 2 & 0x88888888)) & filters << 1;
9645.notraw:
9646.  if (flip == UINT_MAX) flip = tiff_flip;
9647.  if (flip == UINT_MAX) flip = 0;
9648.}
9649.
9650.#ifndef NO_LCMS
9651.void CLASS apply_profile (const char *input, const char *output)
9652.{
9653.  char *prof;
9654.  cmsHPROFILE hInProfile=0, hOutProfile=0;
9655.  cmsHTRANSFORM hTransform;
9656.  FILE *fp;
9657.  unsigned size;
9658.
9659.  if (strcmp (input, "embed"))
9660.    hInProfile = cmsOpenProfileFromFile (input, "r");
9661.  else if (profile_length) {
9662.    prof = (char *) malloc (profile_length);
9663.    merror (prof, "apply_profile()");
9664.    fseek (ifp, profile_offset, SEEK_SET);
9665.    fread (prof, 1, profile_length, ifp);
9666.    hInProfile = cmsOpenProfileFromMem (prof, profile_length);
9667.    free (prof);
9668.  } else
9669.    fprintf (stderr,_("%s has no embedded profile.\n"), ifname);
9670.  if (!hInProfile) return;
9671.  if (!output)
9672.    hOutProfile = cmsCreate_sRGBProfile();
9673.  else if ((fp = fopen (output, "rb"))) {
9674.    fread (&size, 4, 1, fp);
9675.    fseek (fp, 0, SEEK_SET);
9676.    oprof = (unsigned *) malloc (size = ntohl(size));
9677.    merror (oprof, "apply_profile()");
```

```
9678.    fread (oprof, 1, size, fp);
9679.    fclose (fp);
9680.    if (!(hOutProfile = cmsOpenProfileFromMem (oprof, size))) {
9681.      free (oprof);
9682.      oprof = 0;
9683.    }
9684.  } else
9685.    fprintf (stderr,_("Cannot open file %s!\n"), output);
9686.  if (!hOutProfile) goto quit;
9687.  if (verbose)
9688.    fprintf (stderr,_("Applying color profile...\n"));
9689.  hTransform = cmsCreateTransform (hInProfile, TYPE_RGBA_16,
9690.        hOutProfile, TYPE_RGBA_16, INTENT_PERCEPTUAL, 0);
9691.  cmsDoTransform (hTransform, image, image, width*height);
9692.  raw_color = 1;                /* Don't use rgb_cam with a profile */
9693.  cmsDeleteTransform (hTransform);
9694.  cmsCloseProfile (hOutProfile);
9695.quit:
9696.  cmsCloseProfile (hInProfile);
9697.}
9698.#endif
9699.
9700.void CLASS convert_to_rgb()
9701.{
9702.  int row, col, c, i, j, k;
9703.  ushort *img;
9704.  float out[3], out_cam[3][4];
9705.  double num, inverse[3][3];
9706.  static const double xyzd50_srgb[3][3] =
9707.  { { 0.436083, 0.385083, 0.143055 },
9708.    { 0.222507, 0.716888, 0.060608 },
9709.    { 0.013930, 0.097097, 0.714022 } };
9710.  static const double rgb_rgb[3][3] =
9711.  { { 1,0,0 }, { 0,1,0 }, { 0,0,1 } };
9712.  static const double adobe_rgb[3][3] =
9713.  { { 0.715146, 0.284856, 0.000000 },
9714.    { 0.000000, 1.000000, 0.000000 },
9715.    { 0.000000, 0.041166, 0.958839 } };
9716.  static const double wide_rgb[3][3] =
9717.  { { 0.593087, 0.404710, 0.002206 },
9718.    { 0.095413, 0.843149, 0.061439 },
9719.    { 0.011621, 0.069091, 0.919288 } };
9720.  static const double prophoto_rgb[3][3] =
9721.  { { 0.529317, 0.330092, 0.140588 },
9722.    { 0.098368, 0.873465, 0.028169 },
9723.    { 0.016879, 0.117663, 0.865457 } };
9724.  static const double aces_rgb[3][3] =
9725.  { { 0.432996, 0.375380, 0.189317 },
9726.    { 0.089427, 0.816523, 0.102989 },
9727.    { 0.019165, 0.118150, 0.941914 } };
9728.  static const double (*out_rgb[])[3] =
9729.  { rgb_rgb, adobe_rgb, wide_rgb, prophoto_rgb, xyz_rgb, aces_rgb };
9730.  static const char *name[] =
9731.  { "sRGB", "Adobe RGB (1998)", "WideGamut D65", "ProPhoto D65", "XYZ", "ACES" };
9732.  static const unsigned phead[] =
9733.  { 1024, 0, 0x2100000, 0x6d6e7472, 0x52474220, 0x58595a20, 0, 0, 0,
9734.    0x61637370, 0, 0, 0x6e6f6e65, 0, 0, 0, 0, 0xf6d6, 0x10000, 0xd32d };
9735.  unsigned pbody[] =
9736.  { 10, 0x63707274, 0, 36,      /* cprt */
9737.        0x64657363, 0, 40,      /* desc */
9738.        0x77747074, 0, 20,      /* wtpt */
9739.        0x626b7074, 0, 20,      /* bkpt */
9740.        0x72545243, 0, 14,      /* rTRC */
9741.        0x67545243, 0, 14,      /* gTRC */
9742.        0x62545243, 0, 14,      /* bTRC */
```

```
        0x7258595a, 0, 20,      /* rXYZ */
        0x6758595a, 0, 20,      /* gXYZ */
        0x6258595a, 0, 20 };    /* bXYZ */
  static const unsigned pwhite[] = { 0xf351, 0x10000, 0x116cc };
  unsigned pcurve[] = { 0x63757276, 0, 1, 0x1000000 };

  gamma_curve (gamm[0], gamm[1], 0, 0);
  memcpy (out_cam, rgb_cam, sizeof out_cam);
  raw_color |= colors == 1 || document_mode ||
                output_color < 1 || output_color > 6;
  if (!raw_color) {
    oprof = (unsigned *) calloc (phead[0], 1);
    merror (oprof, "convert_to_rgb()");
    memcpy (oprof, phead, sizeof phead);
    if (output_color == 5) oprof[4] = oprof[5];
    oprof[0] = 132 + 12*pbody[0];
    for (i=0; i < pbody[0]; i++) {
      oprof[oprof[0]/4] = i ? (i > 1 ? 0x58595a20 : 0x64657363) : 0x74657874;
      pbody[i*3+2] = oprof[0];
      oprof[0] += (pbody[i*3+3] + 3) & -4;
    }
    memcpy (oprof+32, pbody, sizeof pbody);
    oprof[pbody[5]/4+2] = strlen(name[output_color-1]) + 1;
    memcpy ((char *)oprof+pbody[8]+8, pwhite, sizeof pwhite);
    pcurve[3] = (short)(256/gamm[5]+0.5) << 16;
    for (i=4; i < 7; i++)
      memcpy ((char *)oprof+pbody[i*3+2], pcurve, sizeof pcurve);
    pseudoinverse ((double (*)[3]) out_rgb[output_color-1], inverse, 3);
    for (i=0; i < 3; i++)
      for (j=0; j < 3; j++) {
        for (num = k=0; k < 3; k++)
          num += xyzd50_srgb[i][k] * inverse[j][k];
        oprof[pbody[j*3+23]/4+i+2] = num * 0x10000 + 0.5;
      }
    for (i=0; i < phead[0]/4; i++)
      oprof[i] = htonl(oprof[i]);
    strcpy ((char *)oprof+pbody[2]+8, "auto-generated by dcraw");
    strcpy ((char *)oprof+pbody[5]+12, name[output_color-1]);
    for (i=0; i < 3; i++)
      for (j=0; j < colors; j++)
        for (out_cam[i][j] = k=0; k < 3; k++)
          out_cam[i][j] += out_rgb[output_color-1][i][k] * rgb_cam[k][j];
  }
  if (verbose)
    fprintf (stderr, raw_color ? _("Building histograms...\n") :
        _("Converting to %s colorspace...\n"), name[output_color-1]);

  memset (histogram, 0, sizeof histogram);
  for (img=image[0], row=0; row < height; row++)
    for (col=0; col < width; col++, img+=4) {
      if (!raw_color) {
        out[0] = out[1] = out[2] = 0;
        FORCC {
          out[0] += out_cam[0][c] * img[c];
          out[1] += out_cam[1][c] * img[c];
          out[2] += out_cam[2][c] * img[c];
        }
        FORC3 img[c] = CLIP((int) out[c]);
      }
      else if (document_mode)
        img[0] = img[fcol(row,col)];
      FORCC histogram[c][img[c] >> 3]++;
    }
  if (colors == 4 && output_color) colors = 3;
  if (document_mode && filters) colors = 1;
```

```
}

void CLASS fuji_rotate()
{
  int i, row, col;
  double step;
  float r, c, fr, fc;
  unsigned ur, uc;
  ushort wide, high, (*img)[4], (*pix)[4];

  if (!fuji_width) return;
  if (verbose)
    fprintf (stderr,_("Rotating image 45 degrees...\n"));
  fuji_width = (fuji_width - 1 + shrink) >> shrink;
  step = sqrt(0.5);
  wide = fuji_width / step;
  high = (height - fuji_width) / step;
  img = (ushort (*)[4]) calloc (high, wide*sizeof *img);
  merror (img, "fuji_rotate()");

  for (row=0; row < high; row++)
    for (col=0; col < wide; col++) {
      ur = r = fuji_width + (row-col)*step;
      uc = c = (row+col)*step;
      if (ur > height-2 || uc > width-2) continue;
      fr = r - ur;
      fc = c - uc;
      pix = image + ur*width + uc;
      for (i=0; i < colors; i++)
        img[row*wide+col][i] =
          (pix[    0][i]*(1-fc) + pix[      1][i]*fc) * (1-fr) +
          (pix[width][i]*(1-fc) + pix[width+1][i]*fc) * fr;
    }
  free (image);
  width  = wide;
  height = high;
  image  = img;
  fuji_width = 0;
}

void CLASS stretch()
{
  ushort newdim, (*img)[4], *pix0, *pix1;
  int row, col, c;
  double rc, frac;

  if (pixel_aspect == 1) return;
  if (verbose) fprintf (stderr,_("Stretching the image...\n"));
  if (pixel_aspect < 1) {
    newdim = height / pixel_aspect + 0.5;
    img = (ushort (*)[4]) calloc (width, newdim*sizeof *img);
    merror (img, "stretch()");
    for (rc=row=0; row < newdim; row++, rc+=pixel_aspect) {
      frac = rc - (c = rc);
      pix0 = pix1 = image[c*width];
      if (c+1 < height) pix1 += width*4;
      for (col=0; col < width; col++, pix0+=4, pix1+=4)
        FORCC img[row*width+col][c] = pix0[c]*(1-frac) + pix1[c]*frac + 0.5;
    }
    height = newdim;
  } else {
    newdim = width * pixel_aspect + 0.5;
    img = (ushort (*)[4]) calloc (height, newdim*sizeof *img);
    merror (img, "stretch()");
    for (rc=col=0; col < newdim; col++, rc+=1/pixel_aspect) {
```

```
9873.       frac = rc - (c = rc);
9874.       pix0 = pix1 = image[c];
9875.       if (c+1 < width) pix1 += 4;
9876.       for (row=0; row < height; row++, pix0+=width*4, pix1+=width*4)
9877.         FORCC img[row*newdim+col][c] = pix0[c]*(1-frac) + pix1[c]*frac + 0.5;
9878.     }
9879.     width = newdim;
9880.   }
9881.   free (image);
9882.   image = img;
9883. }
9884.
9885. int CLASS flip_index (int row, int col)
9886. {
9887.   if (flip & 4) SWAP(row,col);
9888.   if (flip & 2) row = iheight - 1 - row;
9889.   if (flip & 1) col = iwidth  - 1 - col;
9890.   return row * iwidth + col;
9891. }
9892.
9893. struct tiff_tag {
9894.   ushort tag, type;
9895.   int count;
9896.   union { char c[4]; short s[2]; int i; } val;
9897. };
9898.
9899. struct tiff_hdr {
9900.   ushort order, magic;
9901.   int ifd;
9902.   ushort pad, ntag;
9903.   struct tiff_tag tag[23];
9904.   int nextifd;
9905.   ushort pad2, nexif;
9906.   struct tiff_tag exif[4];
9907.   ushort pad3, ngps;
9908.   struct tiff_tag gpst[10];
9909.   short bps[4];
9910.   int rat[10];
9911.   unsigned gps[26];
9912.   char desc[512], make[64], model[64], soft[32], date[20], artist[64];
9913. };
9914.
9915. void CLASS tiff_set (struct tiff_hdr *th, ushort *ntag,
9916.         ushort tag, ushort type, int count, int val)
9917. {
9918.   struct tiff_tag *tt;
9919.   int c;
9920.
9921.   tt = (struct tiff_tag *)(ntag+1) + (*ntag)++;
9922.   tt->val.i = val;
9923.   if (type == 1 && count <= 4)
9924.     FORC(4) tt->val.c[c] = val >> (c << 3);
9925.   else if (type == 2) {
9926.     count = strnlen((char *)th + val, count-1) + 1;
9927.     if (count <= 4)
9928.       FORC(4) tt->val.c[c] = ((char *)th)[val+c];
9929.   } else if (type == 3 && count <= 2)
9930.     FORC(2) tt->val.s[c] = val >> (c << 4);
9931.   tt->count = count;
9932.   tt->type = type;
9933.   tt->tag = tag;
9934. }
9935.
9936. #define TOFF(ptr) ((char *)(&(ptr)) - (char *)th)
9937.
```

```
void CLASS tiff_head (struct tiff_hdr *th, int full)
{
  int c, psize=0;
  struct tm *t;

  memset (th, 0, sizeof *th);
  th->order = htonl(0x4d4d4949) >> 16;
  th->magic = 42;
  th->ifd = 10;
  th->rat[0] = th->rat[2] = 300;
  th->rat[1] = th->rat[3] = 1;
  FORC(6) th->rat[4+c] = 1000000;
  th->rat[4] *= shutter;
  th->rat[6] *= aperture;
  th->rat[8] *= focal_len;
  strncpy (th->desc, desc, 512);
  strncpy (th->make, make, 64);
  strncpy (th->model, model, 64);
  strcpy (th->soft, "dcraw v"DCRAW_VERSION);
  t = localtime (&timestamp);
  sprintf (th->date, "%04d:%02d:%02d %02d:%02d:%02d",
      t->tm_year+1900,t->tm_mon+1,t->tm_mday,t->tm_hour,t->tm_min,t->tm_sec);
  strncpy (th->artist, artist, 64);
  if (full) {
    tiff_set (th, &th->ntag, 254, 4, 1, 0);
    tiff_set (th, &th->ntag, 256, 4, 1, width);
    tiff_set (th, &th->ntag, 257, 4, 1, height);
    tiff_set (th, &th->ntag, 258, 3, colors, output_bps);
    if (colors > 2)
      th->tag[th->ntag-1].val.i = TOFF(th->bps);
    FORC4 th->bps[c] = output_bps;
    tiff_set (th, &th->ntag, 259, 3, 1, 1);
    tiff_set (th, &th->ntag, 262, 3, 1, 1 + (colors > 1));
  }
  tiff_set (th, &th->ntag, 270, 2, 512, TOFF(th->desc));
  tiff_set (th, &th->ntag, 271, 2, 64, TOFF(th->make));
  tiff_set (th, &th->ntag, 272, 2, 64, TOFF(th->model));
  if (full) {
    if (oprof) psize = ntohl(oprof[0]);
    tiff_set (th, &th->ntag, 273, 4, 1, sizeof *th + psize);
    tiff_set (th, &th->ntag, 277, 3, 1, colors);
    tiff_set (th, &th->ntag, 278, 4, 1, height);
    tiff_set (th, &th->ntag, 279, 4, 1, height*width*colors*output_bps/8);
  } else
    tiff_set (th, &th->ntag, 274, 3, 1, "12435867"[flip]-'0');
  tiff_set (th, &th->ntag, 282, 5, 1, TOFF(th->rat[0]));
  tiff_set (th, &th->ntag, 283, 5, 1, TOFF(th->rat[2]));
  tiff_set (th, &th->ntag, 284, 3, 1, 1);
  tiff_set (th, &th->ntag, 296, 3, 1, 2);
  tiff_set (th, &th->ntag, 305, 2, 32, TOFF(th->soft));
  tiff_set (th, &th->ntag, 306, 2, 20, TOFF(th->date));
  tiff_set (th, &th->ntag, 315, 2, 64, TOFF(th->artist));
  tiff_set (th, &th->ntag, 34665, 4, 1, TOFF(th->nexif));
  if (psize) tiff_set (th, &th->ntag, 34675, 7, psize, sizeof *th);
  tiff_set (th, &th->nexif, 33434, 5, 1, TOFF(th->rat[4]));
  tiff_set (th, &th->nexif, 33437, 5, 1, TOFF(th->rat[6]));
  tiff_set (th, &th->nexif, 34855, 3, 1, iso_speed);
  tiff_set (th, &th->nexif, 37386, 5, 1, TOFF(th->rat[8]));
  if (gpsdata[1]) {
    tiff_set (th, &th->ntag, 34853, 4, 1, TOFF(th->ngps));
    tiff_set (th, &th->ngps,  0, 1,  4, 0x202);
    tiff_set (th, &th->ngps,  1, 2,  2, gpsdata[29]);
            tiff_set (th, &th->ngps,  2, 5,  3, TOFF(th->gps[0]));
            tiff_set (th, &th->ngps,  3, 2,  2, gpsdata[30]);
            tiff_set (th, &th->ngps,  4, 5,  3, TOFF(th->gps[6]));
```

```
10003.              tiff_set (th, &th->ngps,  5, 1,  1, gpsdata[31]);
10004.              tiff_set (th, &th->ngps,  6, 5,  1, TOFF(th->gps[18]));
10005.              tiff_set (th, &th->ngps,  7, 5,  3, TOFF(th->gps[12]));
10006.              tiff_set (th, &th->ngps, 18, 2, 12, TOFF(th->gps[20]));
10007.              tiff_set (th, &th->ngps, 29, 2, 12, TOFF(th->gps[23]));
10008.              memcpy (th->gps, gpsdata, sizeof th->gps);
10009.            }
10010.          }
10011.
10012.          void CLASS jpeg_thumb()
10013.          {
10014.            char *thumb;
10015.            ushort exif[5];
10016.            struct tiff_hdr th;
10017.
10018.            thumb = (char *) malloc (thumb_length);
10019.            merror (thumb, "jpeg_thumb()");
10020.            fread (thumb, 1, thumb_length, ifp);
10021.            fputc (0xff, ofp);
10022.            fputc (0xd8, ofp);
10023.            if (strcmp (thumb+6, "Exif")) {
10024.              memcpy (exif, "\xff\xe1  Exif\0\0", 10);
10025.              exif[1] = htons (8 + sizeof th);
10026.              fwrite (exif, 1, sizeof exif, ofp);
10027.              tiff_head (&th, 0);
10028.              fwrite (&th, 1, sizeof th, ofp);
10029.            }
10030.            fwrite (thumb+2, 1, thumb_length-2, ofp);
10031.            free (thumb);
10032.          }
10033.
10034.          void CLASS write_ppm_tiff()
10035.          {
10036.            struct tiff_hdr th;
10037.            uchar *ppm;
10038.            ushort *ppm2;
10039.            int c, row, col, soff, rstep, cstep;
10040.            int perc, val, total, white=0x2000;
10041.
10042.            perc = width * height * 0.01;         /* 99th percentile white level */
10043.            if (fuji_width) perc /= 2;
10044.            if (!((highlight & ~2) || no_auto_bright))
10045.              for (white=c=0; c < colors; c++) {
10046.                for (val=0x2000, total=0; --val > 32; )
10047.                  if ((total += histogram[c][val]) > perc) break;
10048.                if (white < val) white = val;
10049.              }
10050.            gamma_curve (gamm[0], gamm[1], 2, (white << 3)/bright);
10051.            iheight = height;
10052.            iwidth  = width;
10053.            if (flip & 4) SWAP(height,width);
10054.            ppm = (uchar *) calloc (width, colors*output_bps/8);
10055.            ppm2 = (ushort *) ppm;
10056.            merror (ppm, "write_ppm_tiff()");
10057.            if (output_tiff) {
10058.              tiff_head (&th, 1);
10059.              fwrite (&th, sizeof th, 1, ofp);
10060.              if (oprof)
10061.                fwrite (oprof, ntohl(oprof[0]), 1, ofp);
10062.            } else if (colors > 3)
10063.              fprintf (ofp,
10064.                "P7\nWIDTH %d\nHEIGHT %d\nDEPTH %d\nMAXVAL %d\nTUPLTYPE %s\nENDHDR\n",
10065.                  width, height, colors, (1 << output_bps)-1, cdesc);
10066.            else
10067.              fprintf (ofp, "P%d\n%d %d\n%d\n",
```

```
10068.                  colors/2+5, width, height, (1 << output_bps)-1);
10069.            soff  = flip_index (0, 0);
10070.            cstep = flip_index (0, 1) - soff;
10071.            rstep = flip_index (1, 0) - flip_index (0, width);
10072.            for (row=0; row < height; row++, soff += rstep) {
10073.              for (col=0; col < width; col++, soff += cstep)
10074.                if (output_bps == 8)
10075.                     FORCC ppm [col*colors+c] = curve[image[soff][c]] >> 8;
10076.                else FORCC ppm2[col*colors+c] = curve[image[soff][c]];
10077.              if (output_bps == 16 && !output_tiff && htons(0x55aa) != 0x55aa)
10078.                swab (ppm2, ppm2, width*colors*2);
10079.              fwrite (ppm, colors*output_bps/8, width, ofp);
10080.            }
10081.            free (ppm);
10082.          }
10083.
10084.          int CLASS main (int argc, const char **argv)
10085.          {
10086.            int arg, status=0, quality, i, c;
10087.            int timestamp_only=0, thumbnail_only=0, identify_only=0;
10088.            int user_qual=-1, user_black=-1, user_sat=-1, user_flip=-1;
10089.            int use_fuji_rotate=1, write_to_stdout=0, read_from_stdin=0;
10090.            const char *sp, *bpfile=0, *dark_frame=0, *write_ext;
10091.            char opm, opt, *ofname, *cp;
10092.            struct utimbuf ut;
10093.          #ifndef NO_LCMS
10094.            const char *cam_profile=0, *out_profile=0;
10095.          #endif
10096.
10097.          #ifndef LOCALTIME
10098.            putenv ((char *) "TZ=UTC");
10099.          #endif
10100.          #ifdef LOCALEDIR
10101.            setlocale (LC_CTYPE, "");
10102.            setlocale (LC_MESSAGES, "");
10103.            bindtextdomain ("dcraw", LOCALEDIR);
10104.            textdomain ("dcraw");
10105.          #endif
10106.
10107.            if (argc == 1) {
10108.              printf(_("\nRaw photo decoder \"dcraw\" v%s"), DCRAW_VERSION);
10109.              printf(_("\nby Dave Coffin, dcoffin a cybercom o net\n"));
10110.              printf(_("\nUsage:  %s [OPTION]... [FILE]...\n\n"), argv[0]);
10111.              puts(_("-v        Print verbose messages"));
10112.              puts(_("-c        Write image data to standard output"));
10113.              puts(_("-e        Extract embedded thumbnail image"));
10114.              puts(_("-i        Identify files without decoding them"));
10115.              puts(_("-i -v     Identify files and show metadata"));
10116.              puts(_("-z        Change file dates to camera timestamp"));
10117.              puts(_("-w        Use camera white balance, if possible"));
10118.              puts(_("-a        Average the whole image for white balance"));
10119.              puts(_("-A <x y w h> Average a grey box for white balance"));
10120.              puts(_("-r <r g b g> Set custom white balance"));
10121.              puts(_("+M/-M     Use/don't use an embedded color matrix"));
10122.              puts(_("-C <r b>  Correct chromatic aberration"));
10123.              puts(_("-P <file> Fix the dead pixels listed in this file"));
10124.              puts(_("-K <file> Subtract dark frame (16-bit raw PGM)"));
10125.              puts(_("-k <num>  Set the darkness level"));
10126.              puts(_("-S <num>  Set the saturation level"));
10127.              puts(_("-n <num>  Set threshold for wavelet denoising"));
10128.              puts(_("-H [0-9]  Highlight mode (0=clip, 1=unclip, 2=blend,
      3+=rebuild)"));
10129.              puts(_("-t [0-7]  Flip image (0=none, 3=180, 5=90CCW, 6=90CW)"));
10130.              puts(_("-o [0-6]  Output colorspace
      (raw,sRGB,Adobe,Wide,ProPhoto,XYZ,ACES)"));
```

```
10131.          #ifndef NO_LCMS
10132.              puts(_("-o <file> Apply output ICC profile from file"));
10133.              puts(_("-p <file> Apply camera ICC profile from file or \"embed\""));
10134.          #endif
10135.              puts(_("-d        Document mode (no color, no interpolation)"));
10136.              puts(_("-D        Document mode without scaling (totally raw)"));
10137.              puts(_("-j        Don't stretch or rotate raw pixels"));
10138.              puts(_("-W        Don't automatically brighten the image"));
10139.              puts(_("-b <num>  Adjust brightness (default = 1.0)"));
10140.              puts(_("-g <p ts> Set custom gamma curve (default = 2.222 4.5)"));
10141.              puts(_("-q [0-3]  Set the interpolation quality"));
10142.              puts(_("-h        Half-size color image (twice as fast as \"-q 0\")"));
10143.              puts(_("-f        Interpolate RGGB as four colors"));
10144.              puts(_("-m <num>  Apply a 3x3 median filter to R-G and B-G"));
10145.              puts(_("-s [0..N-1] Select one raw image or \"all\" from each file"));
10146.              puts(_("-6        Write 16-bit instead of 8-bit"));
10147.              puts(_("-4        Linear 16-bit, same as \"-6 -W -g 1 1\""));
10148.              puts(_("-T        Write TIFF instead of PPM"));
10149.              puts("");
10150.              return 1;
10151.            }
10152.            argv[argc] = "";
10153.            for (arg=1; (((opm = argv[arg][0]) - 2) | 2) == '+'; ) {
10154.              opt = argv[arg++][1];
10155.              if ((cp = (char *) strchr (sp="nbrkStqmHACg", opt)))
10156.                for (i=0; i < "114111111422"[cp-sp]-'0'; i++)
10157.                  if (!isdigit(argv[arg+i][0])) {
10158.                    fprintf (stderr,_("Non-numeric argument to \"-%c\"\n"), opt);
10159.                    return 1;
10160.                  }
10161.              switch (opt) {
10162.                case 'n':  threshold   = atof(argv[arg++]);  break;
10163.                case 'b':  bright      = atof(argv[arg++]);  break;
10164.                case 'r':
10165.                     FORC4 user_mul[c] = atof(argv[arg++]);  break;
10166.                case 'C':  aber[0] = 1 / atof(argv[arg++]);
10167.                           aber[2] = 1 / atof(argv[arg++]);  break;
10168.                case 'g':  gamm[0] =     atof(argv[arg++]);
10169.                           gamm[1] =     atof(argv[arg++]);
10170.                           if (gamm[0]) gamm[0] = 1/gamm[0]; break;
10171.                case 'k':  user_black  = atoi(argv[arg++]);  break;
10172.                case 'S':  user_sat    = atoi(argv[arg++]);  break;
10173.                case 't':  user_flip   = atoi(argv[arg++]);  break;
10174.                case 'q':  user_qual   = atoi(argv[arg++]);  break;
10175.                case 'm':  med_passes  = atoi(argv[arg++]);  break;
10176.                case 'H':  highlight   = atoi(argv[arg++]);  break;
10177.                case 's':
10178.                  shot_select = abs(atoi(argv[arg]));
10179.                  multi_out = !strcmp(argv[arg++],"all");
10180.                  break;
10181.                case 'o':
10182.                  if (isdigit(argv[arg][0]) && !argv[arg][1])
10183.                    output_color = atoi(argv[arg++]);
10184.          #ifndef NO_LCMS
10185.                  else     out_profile = argv[arg++];
10186.                  break;
10187.                case 'p':  cam_profile = argv[arg++];
10188.          #endif
10189.                  break;
10190.                case 'P':  bpfile     = argv[arg++];  break;
10191.                case 'K':  dark_frame = argv[arg++];  break;
10192.                case 'z':  timestamp_only    = 1;  break;
10193.                case 'e':  thumbnail_only    = 1;  break;
10194.                case 'i':  identify_only     = 1;  break;
10195.                case 'c':  write_to_stdout   = 1;  break;
```

```
10196.                case 'v':  verbose           = 1;  break;
10197.                case 'h':  half_size         = 1;  break;
10198.                case 'f':  four_color_rgb    = 1;  break;
10199.                case 'A':  FORC4 greybox[c]  = atoi(argv[arg++]);
10200.                case 'a':  use_auto_wb       = 1;  break;
10201.                case 'w':  use_camera_wb     = 1;  break;
10202.                case 'M':  use_camera_matrix = 3 * (opm == '+');  break;
10203.                case 'I':  read_from_stdin   = 1;  break;
10204.                case 'E':  document_mode++;
10205.                case 'D':  document_mode++;
10206.                case 'd':  document_mode++;
10207.                case 'j':  use_fuji_rotate   = 0;  break;
10208.                case 'W':  no_auto_bright    = 1;  break;
10209.                case 'T':  output_tiff       = 1;  break;
10210.                case '4':  gamm[0] = gamm[1] =
10211.                           no_auto_bright    = 1;
10212.                case '6':  output_bps       = 16;  break;
10213.                default:
10214.                  fprintf (stderr,_("Unknown option \"-%c\".\n"), opt);
10215.                  return 1;
10216.              }
10217.            }
10218.            if (arg == argc) {
10219.              fprintf (stderr,_("No files to process.\n"));
10220.              return 1;
10221.            }
10222.            if (write_to_stdout) {
10223.              if (isatty(1)) {
10224.                fprintf (stderr,_("Will not write an image to the terminal!\n"));
10225.                return 1;
10226.              }
10227.          #if defined(WIN32) || defined(DJGPP) || defined(__CYGWIN__)
10228.              if (setmode(1,O_BINARY) < 0) {
10229.                perror ("setmode()");
10230.                return 1;
10231.              }
10232.          #endif
10233.            }
10234.            for ( ; arg < argc; arg++) {
10235.              status = 1;
10236.              raw_image = 0;
10237.              image = 0;
10238.              oprof = 0;
10239.              meta_data = ofname = 0;
10240.              ofp = stdout;
10241.              if (setjmp (failure)) {
10242.                if (fileno(ifp) > 2) fclose(ifp);
10243.                if (fileno(ofp) > 2) fclose(ofp);
10244.                status = 1;
10245.                goto cleanup;
10246.              }
10247.              ifname = argv[arg];
10248.              if (!(ifp = fopen (ifname, "rb"))) {
10249.                perror (ifname);
10250.                continue;
10251.              }
10252.              status = (identify(),!is_raw);
10253.              if (user_flip >= 0)
10254.                flip = user_flip;
10255.              switch ((flip+3600) % 360) {
10256.                case 270:  flip = 5;  break;
10257.                case 180:  flip = 3;  break;
10258.                case  90:  flip = 6;
10259.              }
10260.              if (timestamp_only) {
```

```
10261.                 if ((status = !timestamp))
10262.                   fprintf (stderr,_("%s has no timestamp.\n"), ifname);
10263.                 else if (identify_only)
10264.                   printf ("%10ld%10d %s\n", (long) timestamp, shot_order, ifname);
10265.                 else {
10266.                   if (verbose)
10267.                     fprintf (stderr,_("%s time set to %d.\n"), ifname, (int) timestamp);
10268.                   ut.actime = ut.modtime = timestamp;
10269.                   utime (ifname, &ut);
10270.                 }
10271.                 goto next;
10272.               }
10273.               write_fun = &CLASS write_ppm_tiff;
10274.               if (thumbnail_only) {
10275.                 if ((status = !thumb_offset)) {
10276.                   fprintf (stderr,_("%s has no thumbnail.\n"), ifname);
10277.                   goto next;
10278.                 } else if (thumb_load_raw) {
10279.                   load_raw = thumb_load_raw;
10280.                   data_offset = thumb_offset;
10281.                   height = thumb_height;
10282.                   width  = thumb_width;
10283.                   filters = 0;
10284.                   colors = 3;
10285.                 } else {
10286.                   fseek (ifp, thumb_offset, SEEK_SET);
10287.                   write_fun = write_thumb;
10288.                   goto thumbnail;
10289.                 }
10290.               }
10291.               if (load_raw == &CLASS kodak_ycbcr_load_raw) {
10292.                 height += height & 1;
10293.                 width  += width  & 1;
10294.               }
10295.               if (identify_only && verbose && make[0]) {
10296.                 printf (_("\nFilename: %s\n"), ifname);
10297.                 printf (_("Timestamp: %s"), ctime(&timestamp));
10298.                 printf (_("Camera: %s %s\n"), make, model);
10299.                 if (artist[0])
10300.                   printf (_("Owner: %s\n"), artist);
10301.                 if (dng_version) {
10302.                   printf (_("DNG Version: "));
10303.                   for (i=24; i >= 0; i -= 8)
10304.                     printf ("%d%c", dng_version >> i & 255, i ? '.':'\n');
10305.                 }
10306.                 printf (_("ISO speed: %d\n"), (int) iso_speed);
10307.                 printf (_("Shutter: "));
10308.                 if (shutter > 0 && shutter < 1)
10309.                   shutter = (printf ("1/"), 1 / shutter);
10310.                 printf (_("%0.1f sec\n"), shutter);
10311.                 printf (_("Aperture: f/%0.1f\n"), aperture);
10312.                 printf (_("Focal length: %0.1f mm\n"), focal_len);
10313.                 printf (_("Embedded ICC profile: %s\n"), profile_length ?
      _("yes"):_("no"));
10314.                 printf (_("Number of raw images: %d\n"), is_raw);
10315.                 if (pixel_aspect != 1)
10316.                   printf (_("Pixel Aspect Ratio: %0.6f\n"), pixel_aspect);
10317.                 if (thumb_offset)
10318.                   printf (_("Thumb size:  %4d x %d\n"), thumb_width, thumb_height);
10319.                 printf (_("Full size:   %4d x %d\n"), raw_width, raw_height);
10320.               } else if (!is_raw)
10321.                 fprintf (stderr,_("Cannot decode file %s\n"), ifname);
10322.               if (!is_raw) goto next;
10323.               shrink = filters && (half_size || (!identify_only &&
10324.                   (threshold || aber[0] != 1 || aber[2] != 1)));
```

```
10325.              iheight = (height + shrink) >> shrink;
10326.              iwidth  = (width  + shrink) >> shrink;
10327.              if (identify_only) {
10328.                if (verbose) {
10329.                  if (document_mode == 3) {
10330.                    top_margin = left_margin = fuji_width = 0;
10331.                    height = raw_height;
10332.                    width  = raw_width;
10333.                  }
10334.                  iheight = (height + shrink) >> shrink;
10335.                  iwidth  = (width  + shrink) >> shrink;
10336.                  if (use_fuji_rotate) {
10337.                    if (fuji_width) {
10338.                      fuji_width = (fuji_width - 1 + shrink) >> shrink;
10339.                      iwidth = fuji_width / sqrt(0.5);
10340.                      iheight = (iheight - fuji_width) / sqrt(0.5);
10341.                    } else {
10342.                      if (pixel_aspect < 1) iheight = iheight / pixel_aspect + 0.5;
10343.                      if (pixel_aspect > 1) iwidth  = iwidth  * pixel_aspect + 0.5;
10344.                    }
10345.                  }
10346.                  if (flip & 4)
10347.                    SWAP(iheight,iwidth);
10348.                  printf (_("Image size:  %4d x %d\n"), width, height);
10349.                  printf (_("Output size: %4d x %d\n"), iwidth, iheight);
10350.                  printf (_("Raw colors: %d"), colors);
10351.                  if (filters) {
10352.                    int fhigh = 2, fwide = 2;
10353.                    if ((filters ^ (filters >>  8)) & 0xff)   fhigh = 4;
10354.                    if ((filters ^ (filters >> 16)) & 0xffff) fhigh = 8;
10355.                    if (filters == 1) fhigh = fwide = 16;
10356.                    if (filters == 9) fhigh = fwide = 6;
10357.                    printf (_("\nFilter pattern: "));
10358.                    for (i=0; i < fhigh; i++)
10359.                      for (c = i && putchar('/') && 0; c < fwide; c++)
10360.                        putchar (cdesc[fcol(i,c)]);
10361.                  }
10362.                  printf (_("\nDaylight multipliers:"));
10363.                  FORCC printf (" %f", pre_mul[c]);
10364.                  if (cam_mul[0] > 0) {
10365.                    printf (_("\nCamera multipliers:"));
10366.                    FORC4 printf (" %f", cam_mul[c]);
10367.                  }
10368.                  putchar ('\n');
10369.                } else
10370.                  printf (_("%s is a %s %s image.\n"), ifname, make, model);
10371.          next:
10372.              fclose(ifp);
10373.              continue;
10374.            }
10375.            if (meta_length) {
10376.              meta_data = (char *) malloc (meta_length);
10377.              merror (meta_data, "main()");
10378.            }
10379.            if (filters || colors == 1) {
10380.              raw_image = (ushort *) calloc ((raw_height+7), raw_width*2);
10381.              merror (raw_image, "main()");
10382.            } else {
10383.              image = (ushort (*)[4]) calloc (iheight, iwidth*sizeof *image);
10384.              merror (image, "main()");
10385.            }
10386.            if (verbose)
10387.              fprintf (stderr,_("Loading %s %s image from %s ...\n"),
10388.                make, model, ifname);
10389.            if (shot_select >= is_raw)
```

```
10390.                fprintf (stderr,_("%s: \"-s %d\" requests a nonexistent image!\n"),
10391.                  ifname, shot_select);
10392.              fseeko (ifp, data_offset, SEEK_SET);
10393.              if (raw_image && read_from_stdin)
10394.                fread (raw_image, 2, raw_height*raw_width, stdin);
10395.              else (*load_raw)();
10396.              if (document_mode == 3) {
10397.                top_margin = left_margin = fuji_width = 0;
10398.                height = raw_height;
10399.                width  = raw_width;
10400.              }
10401.              iheight = (height + shrink) >> shrink;
10402.              iwidth  = (width  + shrink) >> shrink;
10403.              if (raw_image) {
10404.                image = (ushort (*)[4]) calloc (iheight, iwidth*sizeof *image);
10405.                merror (image, "main()");
10406.                crop_masked_pixels();
10407.                free (raw_image);
10408.              }
10409.              if (zero_is_bad) remove_zeroes();
10410.              bad_pixels (bpfile);
10411.              if (dark_frame) subtract (dark_frame);
10412.              quality = 2 + !fuji_width;
10413.              if (user_qual >= 0) quality = user_qual;
10414.              i = cblack[3];
10415.              FORC3 if (i > cblack[c]) i = cblack[c];
10416.              FORC4 cblack[c] -= i;
10417.              black += i;
10418.              i = cblack[6];
10419.              FORC (cblack[4] * cblack[5])
10420.                if (i > cblack[6+c]) i = cblack[6+c];
10421.              FORC (cblack[4] * cblack[5])
10422.                cblack[6+c] -= i;
10423.              black += i;
10424.              if (user_black >= 0) black = user_black;
10425.              FORC4 cblack[c] += black;
10426.              if (user_sat > 0) maximum = user_sat;
10427.          #ifdef COLORCHECK
10428.              colorcheck();
10429.          #endif
10430.              if (is_foveon) {
10431.                if (document_mode || load_raw == &CLASS foveon_dp_load_raw) {
10432.                  for (i=0; i < height*width*4; i++)
10433.                    if ((short) image[0][i] < 0) image[0][i] = 0;
10434.                } else foveon_interpolate();
10435.              } else if (document_mode < 2)
10436.                scale_colors();
10437.              pre_interpolate();
10438.              if (filters && !document_mode) {
10439.                if (quality == 0)
10440.                  lin_interpolate();
10441.                else if (quality == 1 || colors > 3)
10442.                  vng_interpolate();
10443.                else if (quality == 2 && filters > 1000)
10444.                  ppg_interpolate();
10445.                else if (filters == 9)
10446.                  xtrans_interpolate (quality*2-3);
10447.                else
10448.                  ahd_interpolate();
10449.              }
10450.              if (mix_green)
10451.                for (colors=3, i=0; i < height*width; i++)
10452.                  image[i][1] = (image[i][1] + image[i][3]) >> 1;
10453.              if (!is_foveon && colors == 3) median_filter();
10454.              if (!is_foveon && highlight == 2) blend_highlights();
```

```
10455.              if (!is_foveon && highlight > 2) recover_highlights();
10456.              if (use_fuji_rotate) fuji_rotate();
10457.          #ifndef NO_LCMS
10458.              if (cam_profile) apply_profile (cam_profile, out_profile);
10459.          #endif
10460.              convert_to_rgb();
10461.              if (use_fuji_rotate) stretch();
10462.          thumbnail:
10463.              if (write_fun == &CLASS jpeg_thumb)
10464.                write_ext = ".jpg";
10465.              else if (output_tiff && write_fun == &CLASS write_ppm_tiff)
10466.                write_ext = ".tiff";
10467.              else
10468.                write_ext = ".pgm\0.ppm\0.ppm\0.pam" + colors*5-5;
10469.              ofname = (char *) malloc (strlen(ifname) + 64);
10470.              merror (ofname, "main()");
10471.              if (write_to_stdout)
10472.                strcpy (ofname,_("standard output"));
10473.              else {
10474.                strcpy (ofname, ifname);
10475.                if ((cp = strrchr (ofname, '.'))) *cp = 0;
10476.                if (multi_out)
10477.                  sprintf (ofname+strlen(ofname), "_%0*d",
10478.                          snprintf(0,0,"%d",is_raw-1), shot_select);
10479.                if (thumbnail_only)
10480.                  strcat (ofname, ".thumb");
10481.                strcat (ofname, write_ext);
10482.                ofp = fopen (ofname, "wb");
10483.                if (!ofp) {
10484.                  status = 1;
10485.                  perror (ofname);
10486.                  goto cleanup;
10487.                }
10488.              }
10489.              if (verbose)
10490.                fprintf (stderr,_("Writing data to %s ...\n"), ofname);
10491.              (*write_fun)();
10492.              fclose(ifp);
10493.              if (ofp != stdout) fclose(ofp);
10494.          cleanup:
10495.              if (meta_data) free (meta_data);
10496.              if (ofname) free (ofname);
10497.              if (oprof) free (oprof);
10498.              if (image) free (image);
10499.              if (multi_out) {
10500.                if (++shot_select < is_raw) arg--;
10501.                else shot_select = 0;
10502.              }
10503.            }
10504.            return status;
10505.          }
```