Source Code (Use browser search to find items of interest.)

[Class Index]

---

kfax'printruns() (./kdegraphics/kfax/fax2ps.c:58)

```
printruns(unsigned char* buf, uint16* runs, uint16* erun, uint32 lastx)
{
    static struct {
        char white, black;
        short width;
    } WBarr[] = {
        { 'd', 'n', 512 }, { 'e', 'o', 256 }, { 'f', 'p', 128 },
        { 'g', 'q',  64 }, { 'h', 'r',  32 }, { 'i', 's',  16 },
        { 'j', 't',   8 }, { 'k', 'u',   4 }, { 'l', 'v',   2 },
        { 'm', 'w',   1 }
    };
    static char* svalue =
        " !\"#$&'*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_`abc";
    int colormode = 1;          /* 0 for white, 1 for black */
    int runlength = 0;
    int n = maxline;
    int x = 0;
    int l;

    (void) buf;
    fprintf(mypsoutputfile,"%d m(", row++);
    while (runs < erun) {
        if (!runlength) {
            colormode ^= 1;
            runlength = *runs++;
            if (x+runlength > lastx)
                runlength = runs[-1] = lastx-x;
            x += runlength;
            if (!colormode && runs == erun)
                break;          /* don't bother printing the final white run */
        }
        /*
         * If a runlength is greater than 6 pixels, then spit out
         * black or white characters until the runlength drops to
         * 6 or less.  Once a runlength is <= 6, then combine black
         * and white runlengths until a 6-pixel pattern is obtained.
         * Then write out the special character.  Six-pixel patterns
         * were selected since 64 patterns is the largest power of
         * two less than the 92 "easily printable" PostScript
         * characters (i.e., no escape codes or octal chars).
         */
        l = 0;
        while (runlength > 6) { /* Run is greater than six... */
            if (runlength >= WBarr[l].width) {
                if (n == 0) {
                    putc('\n',mypsoutputfile);
                    n = maxline;
                }
                putc(colormode ? WBarr[l].black : WBarr[l].white,mypsoutputfile)
                runlength -= WBarr[l].width;
            } else
                l++;
```

```
        }
        while (runlength > 0 && runlength <= 6) {
            int bitsleft = 6;
            int t = 0;
            while (bitsleft) {
                if (runlength <= bitsleft) {
                    if (colormode)
                        t |= ((1 << runlength)-1) << (bitsleft-runlength);
                    bitsleft -= runlength;
                    runlength = 0;
                    if (bitsleft) {
                        if (runs >= erun)
                            break;
                        colormode ^= 1;
                        runlength = *runs++;
                        if (x+runlength > lastx)
                            runlength = runs[-1] = lastx-x;
                        x += runlength;
                    }
                } else {                /* runlength exceeds bits left */
                    if (colormode)
                        t |= ((1 << bitsleft)-1);
                    runlength -= bitsleft;
                    bitsleft = 0;
                }
            }
            if (n == 0) {
                putc('\n',mypsoutputfile);
                n = maxline;
            }
            putc(svalue[t],mypsoutputfile), n--;
        }
    }
    fprintf(mypsoutputfile,")s\n");
}

void
```

---

kfax'printTIF() (./kdegraphics/kfax/fax2ps.c:147)

```
printTIF(TIFF* tif, int pageNumber)
{
    uint32 w, h;
    uint16 unit;
    float xres, yres;
    tstrip_t s, ns;

    TIFFGetField(tif, TIFFTAG_IMAGELENGTH, &h);
    TIFFGetField(tif, TIFFTAG_IMAGEWIDTH, &w);
    if (!TIFFGetField(tif, TIFFTAG_XRESOLUTION, &xres)) {
        TIFFWarning(TIFFFileName(tif),
            "No x-resolution, assuming %g dpi", defxres);
        xres = defxres;
    }
    if (!TIFFGetField(tif, TIFFTAG_YRESOLUTION, &yres)) {
        TIFFWarning(TIFFFileName(tif),
            "No y-resolution, assuming %g lpi", defyres);
        yres = defyres;                                 /* XXX */
```

```
    }
    if (TIFFGetField(tif, TIFFTAG_RESOLUTIONUNIT, &unit) &&
      unit == RESUNIT_CENTIMETER) {
        xres *= 25.4;
        yres *= 25.4;
    }

    fprintf(mypsoutputfile,"%%%%Page: \"%d\" %d\n", pageNumber, pageNumber);
    fprintf(mypsoutputfile,"gsave\n");
    if (scaleToPage) {
        float yscale = pageHeight / (h/yres);
        float xscale = pageWidth / (w/xres);
        fprintf(mypsoutputfile,"%d %d translate\n",
               (int) (((basePageWidth - pageWidth) * points) * half),
               (int)((yscale*(h/yres)*points) +
               (basePageHeight - pageHeight) * points * half)  );
        fprintf(mypsoutputfile,"%g %g scale\n", (72.*xscale)/xres, -(72.*yscale)
    } else {
        fprintf(mypsoutputfile,"%d %d translate\n",
               (int) ((basePageWidth - pageWidth) * points * half),
               (int)((72.*h/yres) +
               (basePageHeight - pageHeight) * points * half) );
        fprintf(mypsoutputfile,"%g %g scale\n", 72./xres, -72./yres);
    }
    fprintf(mypsoutputfile,"0 setgray\n");
    TIFFSetField(tif, TIFFTAG_FAXFILLFUNC, printruns);
    ns = TIFFNumberOfStrips(tif);
    row = 0;
    for (s = 0; s < ns; s++)
        (void) TIFFReadEncodedStrip(tif, s, (tdata_t) NULL, (tsize_t) -1);
    fprintf(mypsoutputfile,"p\n");
    fprintf(mypsoutputfile,"grestore\n");
    totalPages++;
}
```

---

kfax'findPage() (./kdegraphics/kfax/fax2ps.c:204)

```
findPage(TIFF* tif, int pageNumber)
{
    uint16 pn = (uint16) -1;
    uint16 ptotal = (uint16) -1;
    if (GetPageNumber(tif)) {
        while (pn != pageNumber && TIFFReadDirectory(tif) && GetPageNumber(tif))
            ;
        return (pn == pageNumber);
    } else
        return (TIFFSetDirectory(tif, pageNumber-1));
}

void
```

---

kfax'fax2ps() (./kdegraphics/kfax/fax2ps.c:217)

```
fax2ps(TIFF* tif, int npages, int* pages, const char* filename)
{
```

```
    if (npages > 0) {
        uint16 pn, ptotal;
        int i;

        if (!GetPageNumber(tif))
            fprintf(stderr, "%s: No page numbers, counting directories.\n",
                filename);
        for (i = 0; i < npages; i++) {
            if (findPage(tif, pages[i]))
                printTIF(tif, pages[i]);
            else
                fprintf(stderr, "%s: No page number %d\n", filename, pages[i]);
        }
    } else {
        int pageNumber = 1;
        do
            printTIF(tif, pageNumber++);
        while (TIFFReadDirectory(tif));
    }
}
```

---

kfax'emitFont() (./kdegraphics/kfax/fax2ps.c:248)

```
emitFont(FILE* fd)
{
    static const char* fontPrologue[] = {
        "/newfont 10 dict def newfont begin /FontType 3 def /FontMatrix [1",
        "0 0 1 0 0] def /FontBBox [0 0 512 1] def /Encoding 256 array def",
        "0 1 31{Encoding exch /255 put}for 120 1 255{Encoding exch /255",
        "put}for Encoding 37 /255 put Encoding 40 /255 put Encoding 41 /255",
        "put Encoding 92 /255 put /count 0 def /ls{Encoding exch count 3",
        "string cvs cvn put /count count 1 add def}def 32 1 36{ls}for",
        "38 1 39{ls}for 42 1 91{ls}for 93 1 99{ls}for /count 100",
        "def 100 1 119{ls}for /CharDict 5 dict def CharDict begin /white",
        "{dup 255 eq{pop}{1 dict begin 100 sub neg 512 exch bitshift",
        "/cw exch def cw 0 0 0 cw 1 setcachedevice end}ifelse}def /black",
        "{dup 255 eq{pop}{1 dict begin 110 sub neg 512 exch bitshift",
        "/cw exch def cw 0 0 0 cw 1 setcachedevice 0 0 moveto cw 0 rlineto",
        "0 1 rlineto cw neg 0 rlineto closepath fill end}ifelse}def /numbuild",
        "{dup 255 eq{pop}{6 0 0 0 6 1 setcachedevice 0 1 5{0 moveto",
        "dup 32 and 32 eq{1 0 rlineto 0 1 rlineto -1 0 rlineto closepath",
        "fill newpath}if 1 bitshift}for pop}ifelse}def /.notdef {}",
        "def /255 {}def end /BuildChar{exch begin dup 110 ge{Encoding",
        "exch get 3 string cvs cvi CharDict /black get}{dup 100 ge {Encoding",
        "exch get 3 string cvs cvi CharDict /white get}{Encoding exch get",
        "3 string cvs cvi CharDict /numbuild get}ifelse}ifelse exec end",
        "}def end /Bitfont newfont definefont 1 scalefont setfont",
        NULL
    };
    int i;
    for (i = 0; fontPrologue[i] != NULL; i++)
        fprintf(fd, "%s\n", fontPrologue[i]);
}

static int
```

---

kfax'pcompar() (./kdegraphics/kfax/fax2ps.c:280)

```
pcompar(const void* va, const void* vb)
{
    const int* pa = (const int*) va;
    const int* pb = (const int*) vb;
    return (*pa - *pb);
}
```

---

kfax'fax2psmain() (./kdegraphics/kfax/fax2ps.c:291)

```
fax2psmain(const char* faxtiff_file,  FILE* psoutput,float width, float height, :
{

    int c, pageNumber;
    int* pages = 0, npages = 0;
    int dowarnings = 0;         /* if 1, enable library warnings */
    time_t t;
    TIFF* tif;

    mypsoutputfile = psoutput;
    pageHeight = height;
    pageWidth = width;
    scaleToPage = scale;

/*printf("Width %f Height %f\n",width,height);*/

    /*      pageHeight = atof(optarg);*/
    /*      scaleToPage = 1;*/
    /*      pageWidth = atof(optarg);*/

    /*      pageNumber = atoi(optarg);
            if (pageNumber < 1) {
                fprintf(stderr, "%s: Invalid page number (must be > 0).\n",
                    optarg);
                usage(-1);
            }
            if (pages)
                pages = (int*) realloc((char*) pages, (npages+1)*sizeof (int));
            else
                pages = (int*) malloc(sizeof (int));
            pages[npages++] = pageNumber;
            */

    /*Let's not do any warnings for now*/
    /*Bernd*/


    /*      defxres = atof(optarg);*/
    /*    defyres = atof(optarg);*/
    /*      maxline = atoi(optarg);*/

    if (npages > 0)
        qsort(pages, npages, sizeof (int), pcompar);
    if (!dowarnings)
        TIFFSetWarningHandler(0);
```

```
    fprintf(psoutput,"%%!PS-Adobe-3.0\n");
    fprintf(psoutput,"%%%%Creator: kfax Copyright 1997 Bernd Johannes Wuebben\n"
#ifdef notdef
    fprintf(psoutput,"%%%%Title: %s\n", file);
#endif
    t = time(0);
    fprintf(psoutput,"%%%%CreationDate: %s", ctime(&t));
    fprintf(psoutput,"%%%%Origin: 0 0\n");
    fprintf(psoutput,"%%%%BoundingBox: 0 0 %u %u\n",
        (int)(pageHeight*72), (int)(pageWidth*72));     /* XXX */
    fprintf(psoutput,"%%%%Pages: (atend)\n");
    fprintf(psoutput,"%%%%EndComments\n");
    fprintf(psoutput,"%%%%BeginProlog\n");
    emitFont(psoutput);
    fprintf(psoutput,"/d{bind def}def\n"); /* bind and def proc */
    fprintf(psoutput,"/m{0 exch moveto}d\n");
    fprintf(psoutput,"/s{show}d\n");
    fprintf(psoutput,"/p{showpage}d \n");       /* end page */
    fprintf(psoutput,"%%%%EndProlog\n");


    tif = TIFFOpen(faxtiff_file, "r");
    if (tif) {
      fax2ps(tif, npages, pages, faxtiff_file);
      TIFFClose(tif);
    } else
      fprintf(stderr, "%s: Can not open, or not a TIFF file.\n",
              faxtiff_file);

    fprintf(psoutput,"%%%%Trailer\n");
    fprintf(psoutput,"%%%%Pages: %u\n", totalPages);
    fprintf(psoutput,"%%%%EOF\n");

    return (0);
}

int
```

---

kfax'fax2psmainoriginal() (./kdegraphics/kfax/fax2ps.c:375)

```
fax2psmainoriginal(int argc, char** argv)
{
    extern int optind;
    extern char* optarg;
    int c, pageNumber;
    int* pages = 0, npages = 0;
    int dowarnings = 0;         /* if 1, enable library warnings */
    time_t t;
    TIFF* tif;

    while ((c = getopt(argc, argv, "l:p:x:y:W:H:wS")) != -1)
        switch (c) {
        case 'H':               /* page height */
            pageHeight = atof(optarg);
            break;
        case 'S':               /* scale to page */
```

```
            scaleToPage = 1;
            break;
        case 'W':               /* page width */
            pageWidth = atof(optarg);
            break;
        case 'p':               /* print specific page */
            pageNumber = atoi(optarg);
            if (pageNumber < 1) {
                fprintf(stderr, "%s: Invalid page number (must be > 0).\n",
                    optarg);
                usage(-1);
            }
            if (pages)
                pages = (int*) realloc((char*) pages, (npages+1)*sizeof (int));
            else
                pages = (int*) malloc(sizeof (int));
            pages[npages++] = pageNumber;
            break;
        case 'w':
            dowarnings = 1;
            break;
        case 'x':
            defxres = atof(optarg);
            break;
        case 'y':
            defyres = atof(optarg);
            break;
        case 'l':
            maxline = atoi(optarg);
            break;
        case '?':
            usage(-1);
        }
    if (npages > 0)
        qsort(pages, npages, sizeof (int), pcompar);
    if (!dowarnings)
        TIFFSetWarningHandler(0);
    printf("%%!PS-Adobe-3.0\n");
    printf("%%%%Creator: kfax Copyright 1997 Bernd Johannes Wuebben  wuebben@matl
#ifdef notdef
    printf("%%%%Title: %s\n", file);
#endif
    t = time(0);
    printf("%%%%CreationDate: %s", ctime(&t));
    printf("%%%%Origin: 0 0\n");
    printf("%%%%BoundingBox: 0 0 %u %u\n",
        (int)(pageHeight*72), (int)(pageWidth*72));     /* XXX */
    printf("%%%%Pages: (atend)\n");
    printf("%%%%EndComments\n");
    printf("%%%%BeginProlog\n");
    emitFont(stdout);
    printf("/d{bind def}def\n"); /* bind and def proc */
    printf("/m{0 exch moveto}d\n");
    printf("/s{show}d\n");
    printf("/p{showpage}d \n"); /* end page */
    printf("%%%%EndProlog\n");
    if (optind < argc) {
        do {
            tif = TIFFOpen(argv[optind], "r");
            if (tif) {
                fax2ps(tif, npages, pages, argv[optind]);
```

```
                TIFFClose(tif);
            } else
                fprintf(stderr, "%s: Can not open, or not a TIFF file.\n",
                    argv[optind]);
        } while (++optind < argc);
    } else {
        int n;
        FILE* fd;
        char temp[1024], buf[16*1024];

        strcpy(temp, "/tmp/fax2psXXXXXX");
        (void) mktemp(temp);
        fd = fopen(temp, "w");
        if (fd == NULL) {
            fprintf(stderr, "Could not create temp file \"%s\"\n", temp);
            exit(-2);
        }
        while ((n = read(fileno(stdin), buf, sizeof (buf))) > 0)
            write(fileno(fd), buf, n);
        tif = TIFFOpen(temp, "r");
        unlink(temp);
        if (tif) {
            fax2ps(tif, npages, pages, "<stdin>");
            TIFFClose(tif);
        } else
            fprintf(stderr, "%s: Can not open, or not a TIFF file.\n", temp);
        fclose(fd);
    }
    printf("%%%%Trailer\n");
    printf("%%%%Pages: %u\n", totalPages);
    printf("%%%%EOF\n");

    return (0);
}
```

---

kfax'usage() (./kdegraphics/kfax/fax2ps.c:505)

```
usage(int code)
{
        char buf[BUFSIZ];
        int i;
  return;


        setbuf(stderr, buf);
        for (i = 0; fax2psstuff[i] != NULL; i++)
                fprintf(stderr, "%s\n", fax2psstuff[i]);
        exit(code);
}
```

---

kfax'DummyReadProc() (./kdegraphics/kfax/fax2tiff.c:64)

```
DummyReadProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        (void) fd; (void) buf; (void) size;
        return (0);
}

static tsize_t
```

---

kfax'DummyWriteProc() (./kdegraphics/kfax/fax2tiff.c:71)

```
DummyWriteProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        (void) fd; (void) buf; (void) size;
        return (size);
}

int
```

---

kfax'fax2tiffmain() (./kdegraphics/kfax/fax2tiff.c:78)

```
fax2tiffmain(const char* inputfile, const char* outputfile,int bitorder,int stre
{
  FILE *in;
  TIFF *out = NULL;
  TIFFErrorHandler whandler;
  int compression = COMPRESSION_CCITTFAX3;
  int fillorder = FILLORDER_LSB2MSB;
  uint32 group3options = GROUP3OPT_FILLBITS|GROUP3OPT_2DENCODING;
  int photometric = PHOTOMETRIC_MINISWHITE;
  int mode = FAXMODE_CLASSF;
  int rows;
  int c;
  int pn, npages;

  if(stretchit)
    stretch = 1;
  else
    stretch = 0;


  /* smuggle a descriptor out of the library */
  faxTIFF = TIFFClientOpen("(FakeInput)", "w", (thandle_t) -1,
                         DummyReadProc, DummyWriteProc,
                         NULL, NULL, NULL, NULL, NULL);
  if (faxTIFF == NULL)
    return (EXIT_FAILURE);
  faxTIFF->tif_mode = O_RDONLY;

  TIFFSetField(faxTIFF, TIFFTAG_IMAGEWIDTH,     XSIZE);
  TIFFSetField(faxTIFF, TIFFTAG_SAMPLESPERPIXEL,        1);
  TIFFSetField(faxTIFF, TIFFTAG_BITSPERSAMPLE,  1);
  TIFFSetField(faxTIFF, TIFFTAG_FILLORDER,      FILLORDER_LSB2MSB);
  TIFFSetField(faxTIFF, TIFFTAG_PLANARCONFIG,   PLANARCONFIG_CONTIG);
  TIFFSetField(faxTIFF, TIFFTAG_PHOTOMETRIC,    PHOTOMETRIC_MINISWHITE);
  TIFFSetField(faxTIFF, TIFFTAG_YRESOLUTION,    196.);
```

```
TIFFSetField(faxTIFF, TIFFTAG_RESOLUTIONUNIT, RESUNIT_INCH);

/* NB: this is normally setup when a directory is read */

faxTIFF->tif_scanlinesize = TIFFScanlineSize(faxTIFF);

/* input is 2d-encoded */
/*  if(faxtype == 32)
  TIFFSetField(faxTIFF,TIFFTAG_GROUP3OPTIONS, GROUP3OPT_2DENCODING);
  */
/* input has 0 mean black */
/*            TIFFSetField(faxTIFF,
              TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_MINISBLACK);*/
/* input has lsb-to-msb fillorder */

if(bitorder == 1){

/* input has lsb-to-msb fillorder */
  TIFFSetField(faxTIFF,
               TIFFTAG_FILLORDER, FILLORDER_LSB2MSB);
}
else{

  /*input has msb-to-lsb fillorder*/

  TIFFSetField(faxTIFF,
               TIFFTAG_FILLORDER, FILLORDER_MSB2LSB);
}


/* input resolution */
/*                    TIFFSetField(faxTIFF,
                      TIFFTAG_YRESOLUTION, atof(optarg));*/
/* input has 0 mean white */
/*                    TIFFSetField(faxTIFF,
                      TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_MINISWHITE);*/
/* output-related options */
/* generate 1d-encoded output */
/*                    group3options &= ~GROUP3OPT_2DENCODING;*/
/* generate g4-encoded output */
/*                    compression = COMPRESSION_CCITTFAX4;*/

/* TODO CHECK THIS OUT !!! */

/* generate "classic" g3 format */
/*    mode = FAXMODE_CLASSIC;*/
/* generate Class F format */
/*            mode = FAXMODE_CLASSF;*/
/* output's fillorder is msb-to-lsb */
/*                    fillorder = FILLORDER_MSB2LSB;*/


out = TIFFOpen(outputfile, "w");
if (out == NULL)
  return EXIT_FAILURE;

/* zero pad output scanline EOLs */
/*                    group3options &= ~GROUP3OPT_FILLBITS;*/
/* stretch image by dup'ng scanlines */
/*                    stretch = 1;*/
/* undocumented -- for testing */
```

```
/*                    photometric = PHOTOMETRIC_MINISBLACK;*/
/* undocumented -- for testing */
/*                    compression = COMPRESSION_LZW;*/


faxTIFF->tif_readproc = out->tif_readproc;    /* XXX */
faxTIFF->tif_writeproc = out->tif_writeproc;  /* XXX */
faxTIFF->tif_seekproc = out->tif_seekproc;    /* XXX */
faxTIFF->tif_closeproc = out->tif_closeproc;  /* XXX */
faxTIFF->tif_sizeproc = out->tif_sizeproc;    /* XXX */
faxTIFF->tif_mapproc = out->tif_mapproc;      /* XXX */
faxTIFF->tif_unmapproc = out->tif_unmapproc;  /* XXX */

/*    npages = argc - optind;*/ /* TODO ???????*/

npages = 1;

/* NB: this must be done after directory info is setup */


if(faxtype == 4)
  TIFFSetField(faxTIFF, TIFFTAG_COMPRESSION, COMPRESSION_CCITTFAX4);
else /* faxtype == 32 or 31 */
  TIFFSetField(faxTIFF, TIFFTAG_COMPRESSION, COMPRESSION_CCITTFAX3);

if (faxtype == 32)
  TIFFSetField(faxTIFF, TIFFTAG_GROUP3OPTIONS, GROUP3OPT_2DENCODING);

in = fopen(inputfile, "r" BINMODE);
if (in == NULL) {
  fprintf(stderr,
          "Can not open: %s\n", inputfile);
  return 1;

}

faxTIFF->tif_fd = fileno(in);
faxTIFF->tif_clientdata = (thandle_t) faxTIFF->tif_fd;

/* TODO IS THIS SAFE ??? BERND*/

faxTIFF->tif_name = (char *) inputfile;
TIFFSetField(out, TIFFTAG_IMAGEWIDTH, XSIZE);
TIFFSetField(out, TIFFTAG_BITSPERSAMPLE, 1);
TIFFSetField(out, TIFFTAG_COMPRESSION, compression);
TIFFSetField(out, TIFFTAG_PHOTOMETRIC, photometric);
TIFFSetField(out, TIFFTAG_ORIENTATION, ORIENTATION_TOPLEFT);
TIFFSetField(out, TIFFTAG_SAMPLESPERPIXEL, 1);

if (compression == COMPRESSION_CCITTFAX3) {
  TIFFSetField(out, TIFFTAG_GROUP3OPTIONS, group3options);
  TIFFSetField(out, TIFFTAG_FAXMODE, mode);
}

if (compression == COMPRESSION_CCITTFAX3 ||
    compression == COMPRESSION_CCITTFAX4)
{
  TIFFSetField(out, TIFFTAG_ROWSPERSTRIP, -1L);
} else {
  TIFFSetField(out, TIFFTAG_ROWSPERSTRIP,
               TIFFDefaultStripSize(out, 0));
```

```
  }

  TIFFSetField(out, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);
  TIFFSetField(out, TIFFTAG_FILLORDER, fillorder);
  TIFFSetField(out, TIFFTAG_SOFTWARE, "kfax");
  TIFFSetField(out, TIFFTAG_XRESOLUTION, 204.0);

  if (!stretch) {
    float yres;
    TIFFGetField(faxTIFF, TIFFTAG_YRESOLUTION, &yres);
    TIFFSetField(out, TIFFTAG_YRESOLUTION, yres);
  } else {
    TIFFSetField(out, TIFFTAG_YRESOLUTION, 196.);
  }
  TIFFSetField(out, TIFFTAG_RESOLUTIONUNIT, RESUNIT_INCH);
  TIFFSetField(out, TIFFTAG_PAGENUMBER, pn+1, npages);


  if (!faxt2tiffverbose)
    whandler = TIFFSetWarningHandler(NULL);
  rows = copyFaxFile(faxTIFF, out);
  fclose(in);

  if (!faxt2tiffverbose)
    (void) TIFFSetWarningHandler(whandler);

  TIFFSetField(out, TIFFTAG_IMAGELENGTH, rows);

  if (faxt2tiffverbose) {
    fprintf(stderr, "%s:\n", inputfile);
    fprintf(stderr, "%d rows in input\n", rows);
    fprintf(stderr, "%ld total bad rows\n",
            (long) badfaxlines);
    fprintf(stderr, "%d max consecutive bad rows\n", badfaxrun);
  }
  if (compression == COMPRESSION_CCITTFAX3 &&
      mode == FAXMODE_CLASSF) {
    TIFFSetField(out, TIFFTAG_BADFAXLINES, badfaxlines);
    TIFFSetField(out, TIFFTAG_CLEANFAXDATA, badfaxlines ?
                 CLEANFAXDATA_REGENERATED : CLEANFAXDATA_CLEAN);
    TIFFSetField(out, TIFFTAG_CONSECUTIVEBADFAXLINES, badfaxrun);
  }
  TIFFWriteDirectory(out);

  TIFFClose(out);
  return (EXIT_SUCCESS);
}


int
```

---

kfax'copyFaxFile() (./kdegraphics/kfax/fax2tiff.c:285)

```
copyFaxFile(TIFF* tifin, TIFF* tifout)
{
        uint32 row;
        uint16 badrun;
        int ok;
```

```
        tifin->tif_rawdatasize = TIFFGetFileSize(tifin);
        tifin->tif_rawdata = _TIFFmalloc(tifin->tif_rawdatasize);
        if (!ReadOK(tifin, tifin->tif_rawdata, tifin->tif_rawdatasize)) {
                _TIFFfree(tifin->tif_rawdata);
                TIFFError(tifin->tif_name, "%s: Read error at scanline 0");
                return (0);
        }
        tifin->tif_rawcp = tifin->tif_rawdata;
        tifin->tif_rawcc = tifin->tif_rawdatasize;

        (*tifin->tif_setupdecode)(tifin);
        (*tifin->tif_predecode)(tifin, (tsample_t) 0);
        tifin->tif_row = 0;
        badfaxlines = 0;
        badfaxrun = 0;

        _TIFFmemset(refbuf, 0, sizeof (refbuf));
        row = 0;
        badrun = 0;             /* current run of bad lines */
        while (tifin->tif_rawcc > 0) {
                ok = (*tifin->tif_decoderow)(tifin, rowbuf, sizeof (rowbuf), 0);
                if (!ok) {
                        badfaxlines++;
                        badrun++;
                        /* regenerate line from previous good line */
                        _TIFFmemcpy(rowbuf, refbuf, sizeof (rowbuf));
                } else {
                        if (badrun > badfaxrun)
                                badfaxrun = badrun;
                        badrun = 0;
                        _TIFFmemcpy(refbuf, rowbuf, sizeof (rowbuf));
                }
                tifin->tif_row++;

                if (TIFFWriteScanline(tifout, rowbuf, row, 0) < 0) {
                        fprintf(stderr, "%s: Write error at row %ld.\n",
                            tifout->tif_name, (long) row);
                        break;
                }
                row++;
                if (stretch) {
                        if (TIFFWriteScanline(tifout, rowbuf, row, 0) < 0) {
                                fprintf(stderr, "%s: Write error at row %ld.\n",
                                    tifout->tif_name, (long) row);
                                break;
                        }
                        row++;
                }
        }
        if (badrun > badfaxrun)
                badfaxrun = badrun;
        _TIFFfree(tifin->tif_rawdata);
        return (row);
}
```

---

kfax'usage() (./kdegraphics/kfax/fax2tiff.c:369)

```
usage(void)
{
        char buf[BUFSIZ];
        int i;

        setbuf(stderr, buf);
        for (i = 0; fax2tiffstuff[i] != NULL; i++)
                fprintf(stderr, "%s\n", fax2tiffstuff[i]);
        exit(EXIT_FAILURE);
}
```

---

kfax'unexpected() (./kdegraphics/kfax/faxexpand.cpp:324)

```
unexpected(const char *what, int LineNum)
{
    if (verbose)
        fprintf(stderr, "Line %d: Unexpected state in %s\n",
                LineNum, what);
}

/* Expand tiff modified huffman data (g3-1d without EOLs) */
void
```

---

kfax'MHexpand() (./kdegraphics/kfax/faxexpand.cpp:333)

```
MHexpand(struct pagenode *pn, drawfunc df)
{
    int a0;                     /* reference element */
    int lastx = pn->width;      /* copy line width to register */
    t32bits BitAcc;             /* bit accumulator */
    int BitsAvail;              /* # valid bits in BitAcc */
    int RunLength;              /* Length of current run */
    t16bits *sp;                /* pointer into compressed data */
    pixnum *pa;                 /* pointer into new line */
    int EOLcnt;                 /* number of consecutive EOLs */
    int LineNum;                /* line number */
    pixnum *runs;               /* list of run lengths */
    struct tabent *TabEnt;

    sp = pn->data;
    BitAcc = 0;
    BitsAvail = 0;
    lastx = pn->width;
    runs = (pixnum *) xmalloc(lastx * sizeof(pixnum));
    for (LineNum = 0; LineNum < pn->rowsperstrip; ) {
#ifdef DEBUG
        printf("\nBitAcc=%08lX, BitsAvail = %d\n", BitAcc, BitsAvail);
        printf("-------------------- %d\n", LineNum);
        fflush(stdout);
#endif
        RunLength = 0;
        pa = runs;
        a0 = 0;
        EOLcnt = 0;
        if (BitsAvail & ~8)     /* skip to byte boundary */
```

```
            ClrBits(BitsAvail & ~8);
        expand1d();
        if (RunLength)
            SETVAL(0);
        if (a0 != lastx) {
            if (verbose)
                fprintf(stderr, "Line %d: length is %d (expected %d)\n", LineNum
            while (a0 > lastx)
                a0 -= *--pa;
            if (a0 < lastx) {
                if ((pa - runs) & 1)
                    SETVAL(0);
                SETVAL(lastx - a0);
            }
        }
        (*df)(runs, LineNum++, pn);
    }
    free(runs);
}

/* Expand group-3 1-dimensional data */
void
```

---

kfax'g31expand() (./kdegraphics/kfax/faxexpand.cpp:385)

```
g31expand(struct pagenode *pn, drawfunc df)
{
    int a0;                     /* reference element */
    int lastx = pn->width;      /* copy line width to register */
    t32bits BitAcc;             /* bit accumulator */
    int BitsAvail;              /* # valid bits in BitAcc */
    int RunLength;              /* Length of current run */
    t16bits *sp;                /* pointer into compressed data */
    pixnum *pa;                 /* pointer into new line */
    int EOLcnt;                 /* number of consecutive EOLs */
    int LineNum;                /* line number */
    pixnum *runs;               /* list of run lengths */
    struct tabent *TabEnt;

    sp = pn->data;
    BitAcc = 0;
    BitsAvail = 0;
    lastx = pn->width;
    runs = (pixnum *) xmalloc(lastx * sizeof(pixnum));
    EOLcnt = 0;
    for (LineNum = 0; LineNum < pn->rowsperstrip; ) {
#ifdef DEBUG
        printf("\nBitAcc=%08lX, BitsAvail = %d\n", BitAcc, BitsAvail);
        printf("-------------------- %d\n", LineNum);
        fflush(stdout);
#endif
        if (EOLcnt == 0)
            while (!EndOfData(pn)) {
                /* skip over garbage after a coding error */
                NeedBits(11);
                if (GetBits(11) == 0)
                    break;
                ClrBits(1);
```

```
            }
        for (EOLcnt = 1; !EndOfData(pn); EOLcnt++) {
            /* we have seen 11 zeros, which implies EOL,
               skip possible fill bits too */
            while (1) {
                NeedBits(8);
                if (GetBits(8))
                    break;
                ClrBits(8);
            }
            while (GetBits(1) == 0)
                ClrBits(1);
            ClrBits(1);         /* the eol flag */
            NeedBits(11);
            if (GetBits(11))
                break;
            ClrBits(11);
        }
        if (EOLcnt > 1 && EOLcnt != 6 && verbose)
            fprintf(stderr, "Line %d: bad RTC (%d EOLs)\n", LineNum, EOLcnt);
        if (EOLcnt >= 6 || EndOfData(pn)) {
            free(runs);
            return;
        }
        RunLength = 0;
        pa = runs;
        a0 = 0;
        EOLcnt = 0;
        expand1d();
        if (RunLength)
            SETVAL(0);
        if (a0 != lastx) {
            if (verbose)
                fprintf(stderr, "Line %d: length is %d (expected %d)\n", LineNum
            while (a0 > lastx)
                a0 -= *--pa;
            if (a0 < lastx) {
                if ((pa - runs) & 1)
                    SETVAL(0);
                SETVAL(lastx - a0);
            }
        }
        (*df)(runs, LineNum++, pn);
    }
    free(runs);
}

/* Expand group-3 2-dimensional data */
void
```

---

kfax'g32expand() (./kdegraphics/kfax/faxexpand.cpp:467)

```
g32expand(struct pagenode *pn, drawfunc df)
{
    int RunLength;              /* Length of current run */
    int a0;                     /* reference element */
    int b1;                     /* next change on previous line */
    int lastx = pn->width;      /* copy line width to register */
```

```
    pixnum *run0, *run1;        /* run length arrays */
    pixnum *thisrun, *pa, *pb;  /* pointers into runs */
    t16bits *sp;                /* pointer into compressed data */
    t32bits BitAcc;             /* bit accumulator */
    int BitsAvail;              /* # valid bits in BitAcc */
    int EOLcnt;                 /* number of consecutive EOLs */
    int refline = 0;            /* 1D encoded reference line */
    int LineNum;                /* line number */
    struct tabent *TabEnt;

    sp = pn->data;
    BitAcc = 0;
    BitsAvail = 0;
    /* allocate space for 2 runlength arrays */
    run0 = (pixnum *) xmalloc(2 * ((lastx+5)&~1) * sizeof(pixnum));
    run1 = run0 + ((lastx+5)&~1);
    run1[0] = lastx;
    run1[1] = 0;
    EOLcnt = 0;
    for (LineNum = 0; LineNum < pn->rowsperstrip; ) {
#ifdef DEBUG
        printf("\nBitAcc=%08lX, BitsAvail = %d\n", BitAcc, BitsAvail);
        printf("-------------------- %d\n", LineNum);
        fflush(stdout);
#endif
        if (EOLcnt == 0)
            while (!EndOfData(pn)) {
                /* skip over garbage after a coding error */
                NeedBits(11);
                if (GetBits(11) == 0)
                    break;
                ClrBits(1);
            }
        for (EOLcnt = 1; !EndOfData(pn); EOLcnt++) {
            /* we have seen 11 zeros, which implies EOL,
               skip possible fill bits too */
            while (1) {
                NeedBits(8);
                if (GetBits(8))
                    break;
                ClrBits(8);
            }
            while (GetBits(1) == 0)
                ClrBits(1);
            ClrBits(1);         /* the eol flag */
            NeedBits(12);
            refline = GetBits(1); /* 1D / 2D flag */
            ClrBits(1);
            if (GetBits(11))
                break;
            ClrBits(11);
        }
        if (EOLcnt > 1 && EOLcnt != 6 && verbose)
            fprintf(stderr, "Line %d: bad RTC (%d EOLs)\n", LineNum, EOLcnt);
        if (EOLcnt >= 6 || EndOfData(pn)) {
            free(run0);
            return;
        }
        if (LineNum == 0 && refline == 0 && verbose)
            fprintf(stderr, "First line is 2-D encoded\n");
        RunLength = 0;
```

```
        if (LineNum & 1) {
            pa = run1;
            pb = run0;
        }
        else {
            pa = run0;
            pb = run1;
        }
        thisrun = pa;
        EOLcnt = 0;
        a0 = 0;
        b1 = *pb++;

        if (refline) {
            expand1d();
        }
        else {
            expand2d(EOL2);
        }
        if (RunLength)
            SETVAL(0);
        if (a0 != lastx) {
            if (verbose)
                fprintf(stderr, "Line %d: length is %d (expected %d)\n", LineNum
            while (a0 > lastx)
                a0 -= *--pa;
            if (a0 < lastx) {
                if ((pa - run0) & 1)
                    SETVAL(0);
                SETVAL(lastx - a0);
            }
        }
        SETVAL(0);      /* imaginary change at end of line for reference */
        (*df)(thisrun, LineNum++, pn);
    }
    free(run0);
}

/* Redefine the "skip to eol" macro.  We cannot recover from coding
   errors in G4 data */
```

---

kfax'g4expand() (./kdegraphics/kfax/faxexpand.cpp:586)

```
g4expand(struct pagenode *pn, drawfunc df)
{
    int RunLength;              /* Length of current run */
    int a0;                     /* reference element */
    int b1;                     /* next change on previous line */
    int lastx = pn->width;      /* copy line width to register */
    pixnum *run0, *run1;        /* run length arrays */
    pixnum *thisrun, *pa, *pb;  /* pointers into runs */
    t16bits *sp;                /* pointer into compressed data */
    t32bits BitAcc;             /* bit accumulator */
    int BitsAvail;              /* # valid bits in BitAcc */
    int LineNum;                /* line number */
    int EOLcnt;
    struct tabent *TabEnt;
```

```
    sp = pn->data;
    BitAcc = 0;
    BitsAvail = 0;
    /* allocate space for 2 runlength arrays */
    run0 = (pixnum *) xmalloc(2 * ((lastx+5)&~1) * sizeof(pixnum));
    run1 = run0 + ((lastx+5)&~1);
    run1[0] = lastx;            /* initial reference line */
    run1[1] = 0;

    for (LineNum = 0; LineNum < pn->rowsperstrip; ) {
#ifdef DEBUG
        printf("\nBitAcc=%08lX, BitsAvail = %d\n", BitAcc, BitsAvail);
        printf("-------------------- %d\n", LineNum);
        fflush(stdout);
#endif
        RunLength = 0;
        if (LineNum & 1) {
            pa = run1;
            pb = run0;
        }
        else {
            pa = run0;
            pb = run1;
        }
        thisrun = pa;
        a0 = 0;
        b1 = *pb++;
        expand2d(EOFB);
        if (a0 < lastx) {
            if ((pa - run0) & 1)
                SETVAL(0);
            SETVAL(lastx - a0);
        }
        SETVAL(0);      /* imaginary change at end of line for reference */
        (*df)(thisrun, LineNum++, pn);
        continue;
    EOFB:
        NeedBits(13);
        if (GetBits(13) != 0x1001 && verbose)
            fputs("Bad RTC\n", stderr);
        break;
    }
    free(run0);
}
```

---

kfax'G3count() (./kdegraphics/kfax/faxexpand.cpp:703)

```
G3count(struct pagenode *pn, int twoD)
{
    t16bits *p = pn->data;
    t16bits *end = p + pn->length/sizeof(*p);
    int lines = 0;              /* lines seen so far */
    int zeros = 0;              /* number of consecutive zero bits seen */
    int EOLcnt = 0;             /* number of consecutive EOLs seen */
    int empty = 1;              /* empty line */
    int prezeros, postzeros;
```

```
    while (p < end && EOLcnt < 6) {
        t16bits bits = *p++;
        check(bits&255);
        if (twoD && (prezeros + postzeros == 7)) {
            if (postzeros || ((bits & 0x100) == 0))
                zeros--;
        }
        check(bits>>8);
        if (twoD && (prezeros + postzeros == 7)) {
            if (postzeros || ((p < end) && ((*p & 1) == 0)))
                zeros--;
        }
    }
    return lines - EOLcnt;      /* don't count trailing EOLs */
}
```

---

kfax'FillTable() (./kdegraphics/kfax/faxinit.cpp:297)

```
FillTable(struct tabent *T, int Size, struct proto *P, int State)
{
    int limit = 1 << Size;

    while (P->val) {
        int width = P->val & 15;
        int param = P->val >> 4;
        int incr = 1 << width;
        int code;
        for (code = P->code; code < limit; code += incr) {
            struct tabent *E = T+code;
            E->State = State;
            E->Width = width;
            E->Param = param;
        }
        P++;
    }
}

/* initialise the huffman code tables */
void
```

---

kfax'faxinit() (./kdegraphics/kfax/faxinit.cpp:318)

```
faxinit(void)
{
    FillTable(MainTable, 7, Pass, S_Pass);
    FillTable(MainTable, 7, Horiz, S_Horiz);
    FillTable(MainTable, 7, V0, S_V0);
    FillTable(MainTable, 7, VR, S_VR);
    FillTable(MainTable, 7, VL, S_VL);
    FillTable(MainTable, 7, ExtV, S_Ext);
    FillTable(MainTable, 7, EOLV, S_EOL);
    FillTable(WhiteTable, 12, MakeUpW, S_MakeUpW);
    FillTable(WhiteTable, 12, MakeUp, S_MakeUp);
    FillTable(WhiteTable, 12, TermW, S_TermW);
    FillTable(WhiteTable, 12, ExtH, S_Ext);
    FillTable(WhiteTable, 12, EOLH, S_EOL);
```

```
    FillTable(BlackTable, 13, MakeUpB, S_MakeUpB);
    FillTable(BlackTable, 13, MakeUp, S_MakeUp);
    FillTable(BlackTable, 13, TermB, S_TermB);
    FillTable(BlackTable, 13, ExtH, S_Ext);
    FillTable(BlackTable, 13, EOLH, S_EOL);
}
```

---

kfax'notefile() (./kdegraphics/kfax/faxinput.cpp:40)

```
notefile(const char *name, int type)
{
    struct pagenode *newnode = (struct pagenode *) xmalloc(sizeof *newnode);

    *newnode = defaultpage;
    if (firstpage == NULL){
        firstpage = newnode;
        auxpage = firstpage;
    }
    newnode->prev = lastpage;
    newnode->next = NULL;
    if (lastpage != NULL)
        lastpage->next = newnode;
    lastpage = newnode;

    /*printf("Adding new node%ld\n",newnode);*/

    newnode->pathname = (char*) malloc (strlen(name) +1);
    if(!newnode->pathname){
      kfaxerror("Sorry","Out of memory\n");
      exit(1);
    }

    strcpy(newnode->pathname,name);

    newnode->type = type;


    if ((newnode->name = strrchr(newnode->pathname, '/')) != NULL)
        newnode->name++;
    else
        newnode->name = newnode->pathname;

    if (newnode->width == 0)
        newnode->width = 1728;
    if (newnode->vres < 0)
        newnode->vres = !(newnode->name[0] == 'f' && newnode->name[1] == 'n');
    newnode->extra = NULL;

    return newnode;
}

static t32bits
```

---

kfax'get4() (./kdegraphics/kfax/faxinput.cpp:83)

```
get4(unsigned char *p, int endian)
```

```
{
    return endian ? (p[0]<<24)|(p[1]<<16)|(p[2]<<8)|p[3] :
        p[0]|(p[1]<<8)|(p[2]<<16)|(p[3]<<24);
}

static int
```

---

kfax'get2() (./kdegraphics/kfax/faxinput.cpp:90)

```
get2(unsigned char *p, int endian)
{
    return endian ? (p[0]<<8)|p[1] : p[0]|(p[1]<<8);
}

/* generate pagenodes for the images in a tiff file */
int
```

---

kfax'notetiff() (./kdegraphics/kfax/faxinput.cpp:97)

```
notetiff(const char *name)
{

    FILE *tf;
    unsigned char header[8];
    static const char littleTIFF[5] = "\x49\x49\x2a\x00";
    static const char bigTIFF[5] = "\x4d\x4d\x00\x2a";
    int endian;
    t32bits IFDoff;
    struct pagenode *pn = NULL;
    QString str;


    if ((tf = fopen(name, "r")) == NULL) {
        QString mesg = i18n("Unable to open:\n%1\n").arg(name);
        kfaxerror("Sorry", mesg);
        return 0;
    }

    if (fread(header, 8, 1, tf) == 0) {
    nottiff:
        fclose(tf);
        (void) notefile(name,FAX_RAW);
        return 0;
    }
    if (memcmp(header, &littleTIFF, 4) == 0)
        endian = 0;
    else if (memcmp(header, &bigTIFF, 4) == 0)
        endian = 1;
    else
        goto nottiff;
    IFDoff = get4(header+4, endian);
    if (IFDoff & 1)
        goto nottiff;
    do {                        /* for each page */
        unsigned char buf[8];
        unsigned char *dir = NULL , *dp = NULL;
```

```
        int ndirent;
        pixnum iwidth = defaultpage.width ? defaultpage.width : 1728;
        pixnum iheight = defaultpage.height ? defaultpage.height : 2339;
        int inverse = defaultpage.inverse;
        int lsbfirst = 0;
        int t4opt = 0, comp = 0;
        int orient = defaultpage.orient;
        double yres = defaultpage.vres ? 196.0 : 98.0;
        struct strip *strips = NULL;
        unsigned long rowsperstrip = 0;
        t32bits nstrips = 1;

        if (fseek(tf, IFDoff, SEEK_SET) < 0) {
        realbad:
          str.sprintf("             Invalid tiff file:         \n%s\n",name);
          kfaxerror("Sorry",str);
        bad:
            if (strips)
                free(strips);
            if (dir)
                free(dir);
            fclose(tf);
            return 1;
        }
        if (fread(buf, 2, 1, tf) == 0)
            goto realbad;
        ndirent = get2(buf, endian);
        dir = (unsigned char *) xmalloc(12*ndirent+4);
        if (fread(dir, 12*ndirent+4, 1, tf) == 0)
            goto realbad;
        for (dp = dir; ndirent; ndirent--, dp += 12) {
            /* for each directory entry */
            int tag, ftype;
            t32bits count, value = 0;
            tag = get2(dp, endian);
            ftype = get2(dp+2, endian);
            count = get4(dp+4, endian);
            switch(ftype) {     /* value is offset to list if count*size > 4 */
            case 3:             /* short */
                value = get2(dp+8, endian);
                break;
            case 4:             /* long */
                value = get4(dp+8, endian);
                break;
            case 5:             /* offset to rational */
                value = get4(dp+8, endian);
                break;
            }
            switch(tag) {
            case 256:           /* ImageWidth */
                iwidth = value;
                break;
            case 257:           /* ImageLength */
                iheight = value;
                break;
            case 259:           /* Compression */
                comp = value;
                break;
            case 262:           /* PhotometricInterpretation */
                inverse ^= (value == 1);
                break;
```

```
        case 266:           /* FillOrder */
            lsbfirst = (value == 2);
            break;
        case 273:           /* StripOffsets */
            nstrips = count;
            strips = (struct strip *) xmalloc(count * sizeof *strips);
            if (count == 1 || (count == 2 && ftype == 3)) {
                strips[0].offset = value;
                if (count == 2)
                    strips[1].offset = get2(dp+10, endian);
                break;
            }
            if (fseek(tf, value, SEEK_SET) < 0)
                goto realbad;
            for (count = 0; count < nstrips; count++) {
                if (fread(buf, (ftype == 3) ? 2 : 4, 1, tf) == 0)
                    goto realbad;
                strips[count].offset = (ftype == 3) ?
                    get2(buf, endian) : get4(buf, endian);
            }
            break;
        case 274:           /* Orientation */
            switch(value) {
            default:        /* row0 at top,    col0 at left   */
                orient = 0;
                break;
            case 2:         /* row0 at top,    col0 at right  */
                orient = TURN_M;
                break;
            case 3:         /* row0 at bottom, col0 at right  */
                orient = TURN_U;
                break;
            case 4:         /* row0 at bottom, col0 at left   */
                orient = TURN_U|TURN_M;
                break;
            case 5:         /* row0 at left,   col0 at top    */
                orient = TURN_M|TURN_L;
                break;
            case 6:         /* row0 at right,  col0 at top    */
                orient = TURN_U|TURN_L;
                break;
            case 7:         /* row0 at right,  col0 at bottom */
                orient = TURN_U|TURN_M|TURN_L;
                break;
            case 8:         /* row0 at left,   col0 at bottom */
                orient = TURN_L;
                break;
            }
            break;
        case 278:           /* RowsPerStrip */
            rowsperstrip = value;
            break;
        case 279:           /* StripByteCounts */
            if (count != nstrips) {
              str.sprintf("In file %s\nStrpisPerImage tag 273=%ls,tag279=%ld
                          name, nstrips,(long int) count);
              kfaxerror("Message",str);
              goto realbad;
            }
            if (count == 1 || (count == 2 && ftype == 3)) {
                strips[0].size = value;
```

```
                if (count == 2)
                    strips[1].size = get2(dp+10, endian);
                break;
            }
            if (fseek(tf, value, SEEK_SET) < 0)
                goto realbad;
            for (count = 0; count < nstrips; count++) {
                if (fread(buf, (ftype == 3) ? 2 : 4, 1, tf) == 0)
                    goto realbad;
                strips[count].size = (ftype == 3) ?
                    get2(buf, endian) : get4(buf, endian);
            }
            break;
        case 283:           /* YResolution */
            if (fseek(tf, value, SEEK_SET) < 0 ||
                fread(buf, 8, 1, tf) == 0)
                goto realbad;
            yres = get4(buf, endian) / get4(buf+4, endian);
            break;
        case 292:           /* T4Options */
            t4opt = value;
            break;
        case 293:           /* T6Options */
            /* later */
            break;
        case 296:           /* ResolutionUnit */
            if (value == 3)
                yres *= 2.54;
            break;
        }
    }
    IFDoff = get4(dp, endian);
    free(dir);
    dir = NULL;
    if (comp < 2 || comp > 4) {
      kfaxerror("Sorry","This version can only handle Fax files\n");
        goto bad;
    }
    pn = notefile(name,FAX_TIFF);
    pn->nstrips = nstrips;
    pn->rowsperstrip = nstrips > 1 ? rowsperstrip : iheight;
    pn->strips = strips;
    pn->width = iwidth;
    pn->height = iheight;
    pn->inverse = inverse;
    pn->lsbfirst = lsbfirst;
    pn->orient = orient;
    pn->vres = (yres > 150); /* arbitrary threshold for fine resolution */
    if (comp == 2)
        pn->expander = MHexpand;
    else if (comp == 3)
        pn->expander = (t4opt & 1) ? g32expand : g31expand;
    else
        pn->expander = g4expand;
  } while (IFDoff);
  fclose(tf);
  return 1;
}

/* report error and remove bad file from the list */
static void
```

kfax'badfile() (./kdegraphics/kfax/faxinput.cpp:317)

```
badfile(struct pagenode *pn)
{
    struct pagenode *p;

    if (errno)
        perror(pn->pathname);
    if (pn == firstpage) {
      if (pn->next == NULL){
          kfaxerror("Sorry","             Bad Fax File          ");
          return;
      }
      else{
        firstpage = thispage = firstpage->next;
        firstpage->prev = NULL;
      }
    }
    else
        for (p = firstpage; p; p = p->next)
            if (p->next == pn) {
                thispage = p;
                p->next = pn->next;
                if (pn->next)
                    pn->next->prev = p;
                break;
            }
    if (pn) free(pn);
    pn = NULL;
}

/* rearrange input bits into t16bits lsb-first chunks */
static void
```

kfax'normalize() (./kdegraphics/kfax/faxinput.cpp:348)

```
normalize(struct pagenode *pn, int revbits, int swapbytes, size_t length)
{
    t32bits *p = (t32bits *) pn->data;

    switch ((revbits<<1)|swapbytes) {
    case 0:
        break;
    case 1:
        for ( ; length; length -= 4) {
            t32bits t = *p;
            *p++ = ((t & 0xff00ff00) >> 8) | ((t & 0x00ff00ff) << 8);
        }
        break;
    case 2:
        for ( ; length; length -= 4) {
            t32bits t = *p;
            t = ((t & 0xf0f0f0f0) >> 4) | ((t & 0x0f0f0f0f) << 4);
            t = ((t & 0xcccccccc) >> 2) | ((t & 0x33333333) << 2);
            *p++ = ((t & 0xaaaaaaaa) >> 1) | ((t & 0x55555555) << 1);
```

```
        }
        break;
    case 3:
        for ( ; length; length -= 4) {
            t32bits t = *p;
            t = ((t & 0xff00ff00) >> 8) | ((t & 0x00ff00ff) << 8);
            t = ((t & 0xf0f0f0f0) >> 4) | ((t & 0x0f0f0f0f) << 4);
            t = ((t & 0xcccccccc) >> 2) | ((t & 0x33333333) << 2);
            *p++ = ((t & 0xaaaaaaaa) >> 1) | ((t & 0x55555555) << 1);
        }
    }
}


/* get compressed data into memory */
unsigned char *
```

---

kfax'getstrip() (./kdegraphics/kfax/faxinput.cpp:383)

```
getstrip(struct pagenode *pn, int strip)
{
    int fd;
    size_t offset, roundup;
    struct stat sbuf;
    unsigned char *Data;
    union { t16bits s; unsigned char b[2]; } so;
    QString str;
```

---

kfax'copy() (./kdegraphics/kfax/g3hack.cpp:57)

```
copy(int nlines)
{
    int ibits = 0, imask = 0;   /* input bits and mask */
    int obits = 0;              /* output bits */
    int omask = 0x80;           /* output mask */
    int zeros = 0;              /* number of consecutive zero bits */
    int thisempty = 1;          /* empty line (so far) */
    int empties = 0;            /* number of consecutive EOLs */
    int identcount = 0;         /* number of consecutive identical lines */
    struct {
        char line[BUFSIZ];
        int length;
    } lines[2], *prev, *this, *temp;

    this = &lines[0];
    prev = &lines[1];
    this->length = prev->length = 0;
    while (1) {
        int bit = nxtbit();
        if (bit == -1)
            break;              /* end of file */
        putbit(bit);
        if (bit == 0) {
            zeros++;
            continue;
```

```
        }
        if (zeros < 11) {       /* not eol and not empty */
            zeros = 0;
            thisempty = 0;
            /* Get rid of any accumulated empties.  Should only happen
               for the eol at the beginning of the first line (we
               switch from the |eol data| to the |data eol|
               viewpoint). */
            for ( ; empties; empties--)
                if (fwrite("\0\1", 1, 2, stdout) != 2)
                    break;
            continue;
        }
        /* at end of line */
        zeros = 0;
        omask = 0x80;
        obits = 0;
        if (thisempty) {
            empties++;
            if (empties >= 5)
                break;          /* 6 eols in a row */
            this->length = 0;
            continue;
        }
        thisempty = 1;
        /* at end of non-empty line */
        this->length = (this->length+7)&~7;
        this->line[(this->length-1)>>3] = 1; /* byte-align the eol */
        if (this->length == prev->length &&
            memcmp(this->line, prev->line, this->length>>3) == 0) {
            identcount++;
            this->length = 0;
            continue;
        }
        /* at end of non-matching line */
        for ( ; identcount; identcount--)
            if (fwrite(prev->line, 1, prev->length>>3, stdout) !=
                prev->length>>3)
                break;
        temp = prev;
        prev = this;
        this = temp;
        identcount = 1;
        this->length = 0;
    }
    if (identcount > nlines)
        identcount = nlines;
    for ( ; !ferror(stdout) && identcount; identcount--)
            fwrite(prev->line, 1, prev->length>>3, stdout);
    if (!ferror(stdout) && !thisempty)
            fwrite(this->line, 1, this->length>>3, stdout);
    for ( ; !ferror(stdout) && empties; empties--)
        fwrite("\0\1", 1, 2, stdout);
    if (ferror(stdout)) {
        fprintf(stderr, "%s: write error\n", progname);
        exit(1);
    }
}

int
```

---

kfax'main() (./kdegraphics/kfax/g3hack.cpp:142)

```
main(int argc, char **argv)
{
    int c, err = 0;
    int header = 0;
    int nlines = 10;

    if ((progname = strrchr(argv[0], '/')) == NULL)
        progname = argv[0];
    else
        progname++;
    opterr = 0;
    while ((c = getopt(argc, argv, "h:n:o:v")) != EOF)
        switch (c) {
        case 'h':
            header = atoi(optarg);
            break;
        case 'n':
            nlines = atoi(optarg);
            break;
        case 'o':
            if (freopen(optarg, "w", stdout) == NULL) {
                perror(optarg);
                exit(1);
            }
            break;
        case 'v':
            fprintf(stderr, banner, progname);
            exit(0);
        case '?':
            err++;
        }
    if (err || optind < argc-1) {
        fprintf(stderr, banner, progname);
        fprintf(stderr, usage, progname);
        exit(1);
    }
    if (optind < argc && freopen(argv[optind], "r", stdin) == NULL) {
        perror(argv[optind]);
        exit(1);
    }
    while (header--)
        putchar(getchar());
    copy(nlines);
    exit(0);
}
```

---

kfax'SetupDisplay() (./kdegraphics/kfax/kfax.cpp:1403)

```
void SetupDisplay(){

  if(display_is_setup){
    return;
  }
```

```
display_is_setup = TRUE;

xpos = ypos = ox = oy = 0;
ExpectConfNotify = 1;

/*  XSizeHints size_hints;*/

zfactor = 4; // a power of two
// the original image size is zfactor 0 the next small size (half as large)
// is zfactor 1 etc.

/*
int faxh = Pimage(thispage)->height;
int faxw = Pimage(thispage)->width;

// TODO Let the user choose this in a settings dialog


int i;
for (size_hints.width = faxw, i = 1; i < zfactor; i *= 2)
  size_hints.width = (size_hints.width + 1) /2;

for (size_hints.height = faxh, i = 1; i < zfactor; i *= 2)
  size_hints.height = (size_hints.height + 1) /2;

Win = XCreateSimpleWindow(qtdisplay,qtwin,0,0,
                          size_hints.width,size_hints.height,
                          0,
                          BlackPixel(qtdisplay,XDefaultScreen(qtdisplay)),
                          WhitePixel(qtdisplay,XDefaultScreen(qtdisplay)));

                          */

Win = XCreateSimpleWindow(qtdisplay,qtwin,1,1,
                          1,1,
                          0,
                          BlackPixel(qtdisplay,XDefaultScreen(qtdisplay)),
                          WhitePixel(qtdisplay,XDefaultScreen(qtdisplay)));

PaintGC = XCreateGC(qtdisplay, Win, 0L, (XGCValues *) NULL);
XSetForeground(qtdisplay, PaintGC, BlackPixel(qtdisplay, XDefaultScreen(qtdisp
XSetBackground(qtdisplay, PaintGC, WhitePixel(qtdisplay, XDefaultScreen(qtdisp
XSetFunction(qtdisplay, PaintGC, GXcopy);
WorkCursor = XCreateFontCursor(qtdisplay, XC_watch);
//ReadyCursor = XCreateFontCursor(qtdisplay, XC_plus);
ReadyCursor = XCreateFontCursor(qtdisplay, XC_hand2);
MoveCursor = XCreateFontCursor(qtdisplay, XC_fleur);
LRCursor = XCreateFontCursor(qtdisplay, XC_sb_h_double_arrow);
UDCursor = XCreateFontCursor(qtdisplay, XC_sb_v_double_arrow);

XSelectInput(qtdisplay, Win, Button2MotionMask | ButtonPressMask |
             ButtonReleaseMask | ExposureMask | KeyPressMask |
             SubstructureNotifyMask | LeaveWindowMask | OwnerGrabButtonMask |
             StructureNotifyMask);

XMapRaised(qtdisplay, Win);

XDefineCursor(qtdisplay, Win, ReadyCursor);
XFlush(qtdisplay);
```

```
  for (oz = 0; oz < MAXZOOM; oz++)
    Images[oz] = NULL;


  // setup oz the default zoom factor
  for (oz = 0; oz < MAXZOOM && zfactor > (1 << oz); oz++){
  }


}
```

---

kfax'kfaxerror() (./kdegraphics/kfax/kfax.cpp:2317)

```
void kfaxerror(const QString& title, const QString& error){
  KMessageBox::error(0, error, title);
}
```

---

kfax'setFaxDefaults() (./kdegraphics/kfax/kfax.cpp:2425)

```
void setFaxDefaults(){

  // fop is called in readSettings, so this can't be
  // called after a TopLevel::readSettings()

  if(have_cmd_opt ) // we have commad line options all kfaxrc defaults are
    return;         // overridden

  if(fop.resauto == 1){
    defaultpage.vres = -1;
  }
  else{
    defaultpage.vres = fop.fine;
  }

  if(fop.geomauto == 1){
    defaultpage.width = 0;
    defaultpage.height = 0;
  }
  else{
    defaultpage.width = fop.width;
    defaultpage.height = fop.height;
  }

  if( fop.landscape)
    defaultpage.orient |= TURN_L;

  if(fop.flip)
     defaultpage.orient |= TURN_U;

  defaultpage.inverse =    fop.invert;
  defaultpage.lsbfirst = fop.lsbfirst;
```

```
  if(fop.raw == 2)
    defaultpage.expander = g32expand;
  if(fop.raw == 4)
    defaultpage.expander = g4expand;
  if((fop.raw != 4) && (fop.raw != 2) )
    defaultpage.expander = g31expand;

}
```

---

kfax'main() (./kdegraphics/kfax/kfax.cpp:2470)

```
int main (int argc, char **argv)
{
  toplevel = NULL;

  catchSignals();

  KAboutData aboutData( "kfax", I18N_NOOP("KFax"),
      KFAXVERSION, description, KAboutData::License_GPL,
      "(c) 1997-98 Bernd Johannes Wuebben");
  aboutData.addAuthor( "Bernd Johannes Wuebben", 0, "wuebben@kde.org" );

  KCmdLineArgs::init(argc, argv, &aboutData);

  viewfax_addCmdLineOptions();

  MyApp a;

  qtdisplay = qt_xdisplay();

  viewfaxmain();

  toplevel = new TopLevel();
  toplevel->show ();

  startingup = 1;
  a.processEvents();
  a.flushX();

  startingup = 0;

  faxinit();

  if(!have_no_fax){

    thispage = firstpage;

    toplevel->newPage();
    toplevel->resizeView();
    //TODO : I don't think I need this putImage();
    toplevel->putImage();
  }

  toplevel->uiUpdate();

  return a.exec ();
```

```
}
```

---

kfax'setfaxtitle() (./kdegraphics/kfax/kfax.cpp:2517)

```
void setfaxtitle(const QString& name){

  toplevel->setFaxTitle(name);

}
```

---

kfax'setstatusbarmem() (./kdegraphics/kfax/kfax.cpp:2523)

```
void setstatusbarmem(int mem){

  if(toplevel)
    toplevel->setStatusBarMemField(mem);

}
```

---

kfax'mysighandler() (./kdegraphics/kfax/kfax.cpp:2531)

```
void mysighandler(int ){

  //  printf("signal received %d\n",sig);
  catchSignals(); // reinstall signal handler

}
```

---

kfax'catchSignals() (./kdegraphics/kfax/kfax.cpp:2539)

```
void catchSignals()
{
  /*
        signal(SIGHUP, mysighandler);
        signal(SIGINT, mysighandler);
        signal(SIGTERM, mysighandler);
        signal(SIGCHLD, mysighandler);
        signal(SIGABRT, mysighandler);
        signal(SIGUSR1, mysighandler);
        signal(SIGALRM, mysighandler);
        signal(SIGFPE, mysighandler);
        signal(SIGILL, mysighandler);*/

        signal(SIGPIPE, mysighandler);
```

```
/*      signal(SIGQUIT, mysighandler);
        signal(SIGSEGV, mysighandler);

#ifdef SIGBUS
        signal(SIGBUS, mysighandler);
#endif
#ifdef SIGPOLL
        signal(SIGPOLL, mysighandler);
#endif
#ifdef SIGSYS
        signal(SIGSYS, mysighandler);
#endif
#ifdef SIGTRAP
        signal(SIGTRAP, mysighandler);
#endif
#ifdef SIGVTALRM
        signal(SIGVTALRM, mysighandler);
#endif
#ifdef SIGXCPU
        signal(SIGXCPU, mysighandler);
#endif
#ifdef SIGXFSZ
        signal(SIGXFSZ, mysighandler);
#endif
*/

}
```

---

kfax'parse() (./kdegraphics/kfax/kfax.cpp:2584)

```
void parse(char* buf, char** args){

  while(*buf != '\0'){

    // Strip whitespace. Use nulls, so that the previous argument is terminated
    // automatically.

    while ((*buf == ' ' ) || (*buf == '\t' ) || (*buf == '\n' ) )
      *buf++ ='\0';

    // save the argument
    if(*buf != '\0')
    *args++ = buf;

    while ((*buf != '\0') && (*buf != '\n') && (*buf != '\t') && (*buf != ' '))
      buf++;

  }

  *args ='\0';;

}
```

---

kfax'copyfile() (./kdegraphics/kfax/kfax.cpp:2607)

```
int copyfile(const char* toname,char* fromname){

  char buffer[4*1028];
  int count = 0;
  int count2;

  FILE*  fromfile;
  FILE*  tofile;

  if (QFile::exists(toname)) {
    if(KMessageBox::questionYesNo( 0,
                              i18n("A file with this name already exists\n"
                              "Do you want to overwrite it?\n\n")))
      return 1;
  }

  if((fromfile = fopen(fromname,"r")) == NULL)
    return 0;

  if((tofile =  fopen(toname,"w")) == NULL){
    fclose(fromfile);
    return 0;
  }


  while((count = fread(buffer,sizeof(char),4*1028,fromfile))){
    count2 = fwrite(buffer,sizeof(char),count,tofile);
    if (count2 != count){
      fclose(fromfile);
      fclose(tofile);
      return 0;
    }

  }
  fclose(fromfile);
  fclose(tofile);
  return 1;

}
```

---

kfax'FillTable() (./kdegraphics/kfax/libtiffax/mkg3states.c:314)

```
FillTable(TIFFFaxTabEnt *T, int Size, struct proto *P, int State)
{
    int limit = 1 << Size;

    while (P->val) {
        int width = P->val & 15;
        int param = P->val >> 4;
        int incr = 1 << width;
        int code;
        for (code = P->code; code < limit; code += incr) {
            TIFFFaxTabEnt *E = T+code;
            E->State = State;
            E->Width = width;
```

```
            E->Param = param;
        }
        P++;
    }
}
```

---

kfax'WriteTable() (./kdegraphics/kfax/libtiffax/mkg3states.c:340)

```
WriteTable(FILE* fd, const TIFFFaxTabEnt* T, int Size, const char* name)
{
    int i;
    char* sep;

    fprintf(fd, "%s %s TIFFFaxTabEnt %s[%d] = {",
        storage_class, const_class, name, Size);
    if (packoutput) {
        sep = "\n";
        for (i = 0; i < Size; i++) {
            fprintf(fd, "%s%s%d,%d,%d%s",
                sep, prebrace, T->State, T->Width, T->Param, postbrace);
            if (((i+1) % 12) == 0)
                    sep = ",\n";
            else
                    sep = ",";
            T++;
        }
    } else {
        sep = "\n ";
        for (i = 0; i < Size; i++) {
            fprintf(fd, "%s%s%3d,%3d,%4d%s",
                sep, prebrace, T->State, T->Width, T->Param, postbrace);
            if (((i+1) % 6) == 0)
                    sep = ",\n ";
            else
                    sep = ",";
            T++;
        }
    }
    fprintf(fd, "\n};\n");
}

/* initialise the huffman code tables */
int
```

---

kfax'main() (./kdegraphics/kfax/libtiffax/mkg3states.c:375)

```
main(int argc, char* argv[])
{
    FILE* fd;
    char* outputfile;
    int c;
    extern int optind;
    extern char* optarg;

    while ((c = getopt(argc, argv, "c:s:bp")) != -1)
```

```
        switch (c) {
        case 'c':
            const_class = optarg;
            break;
        case 's':
            storage_class = optarg;
            break;
        case 'p':
            packoutput = 0;
            break;
        case 'b':
            prebrace = "{";
            postbrace = "}";
            break;
        case '?':
            fprintf(stderr,
                "usage: %s [-c const] [-s storage] [-p] [-b] file\n",
                argv[0]);
            return (-1);
        }
    outputfile = optind < argc ? argv[optind] : "g3states.h";
    fd = fopen(outputfile, "w");
    if (fd == NULL) {
        fprintf(stderr, "%s: %s: Cannot create output file.\n",
            argv[0], outputfile);
        return (-2);
    }
    FillTable(MainTable, 7, Pass, S_Pass);
    FillTable(MainTable, 7, Horiz, S_Horiz);
    FillTable(MainTable, 7, V0, S_V0);
    FillTable(MainTable, 7, VR, S_VR);
    FillTable(MainTable, 7, VL, S_VL);
    FillTable(MainTable, 7, Ext, S_Ext);
    FillTable(MainTable, 7, EOLV, S_EOL);
    FillTable(WhiteTable, 12, MakeUpW, S_MakeUpW);
    FillTable(WhiteTable, 12, MakeUp, S_MakeUp);
    FillTable(WhiteTable, 12, TermW, S_TermW);
    FillTable(WhiteTable, 12, EOLH, S_EOL);
    FillTable(BlackTable, 13, MakeUpB, S_MakeUpB);
    FillTable(BlackTable, 13, MakeUp, S_MakeUp);
    FillTable(BlackTable, 13, TermB, S_TermB);
    FillTable(BlackTable, 13, EOLH, S_EOL);

    fprintf(fd, "/* WARNING, this file was automatically generated by the\n");
    fprintf(fd, "    mkg3states program */\n");
    fprintf(fd, "#include \"tiff.h\"\n");
    fprintf(fd, "#include \"tif_fax3.h\"\n");
    WriteTable(fd, MainTable, 128, "TIFFFaxMainTable");
    WriteTable(fd, WhiteTable, 4096, "TIFFFaxWhiteTable");
    WriteTable(fd, BlackTable, 8192, "TIFFFaxBlackTable");
    fclose(fd);
    return (0);
}
```

---

kfax'dumparray() (./kdegraphics/kfax/libtiffax/mkspans.c:34)

```
dumparray(name, runs)
        char *name;
```

```
        unsigned char runs[256];
{
        register int i;
        register char *sep;
        printf("static u_char %s[256] = {\n", name);
        sep = "    ";
        for (i = 0; i < 256; i++) {
                printf("%s%d", sep, runs[i]);
                if (((i + 1) % 16) == 0) {
                        printf(",       /* 0x%02x - 0x%02x */\n", i-15, i);
                        sep = "    ";
                } else
                        sep = ", ";
        }
        printf("\n};\n");
}
```

---

kfax'main() (./kdegraphics/kfax/libtiffax/mkspans.c:53)

```
main()
{
        unsigned char runs[2][256];

        memset(runs[0], 0, 256*sizeof (char));
        memset(runs[1], 0, 256*sizeof (char));
        { register int run, runlen, i;
          runlen = 1;
          for (run = 0x80; run != 0xff; run = (run>>1)|0x80) {
                for (i = run-1; i >= 0; i--) {
                        runs[1][run|i] = runlen;
                        runs[0][(~(run|i)) & 0xff] = runlen;
                }
                runlen++;
          }
          runs[1][0xff] = runs[0][0] = 8;
        }
        dumparray("bruns", runs[0]);
        dumparray("wruns", runs[1]);
}
```

---

kfax'usage() (./kdegraphics/kfax/libtiffax/mkversion.c:41)

```
usage(void)
{
    fprintf(stderr,
        "usage: mkversion [-v version-file] [-a alpha-file] [outfile]\n");
    exit(-1);
}

static FILE*
```

---

kfax'openFile() (./kdegraphics/kfax/libtiffax/mkversion.c:49)

```
openFile(char* filename)
{
    FILE* fd = fopen(filename, "r");
    if (fd == NULL) {
        fprintf(stderr, "mkversion: %s: Could not open for reading.\n",
            filename);
        exit(-1);
    }
    return (fd);
}

int
```

---

kfax'main() (./kdegraphics/kfax/libtiffax/mkversion.c:61)

```
main(int argc, char* argv[])
{
    char* versionFile = "../VERSION";
    char* alphaFile = "../dist/tiff.alpha";
    char version[128];
    char alpha[128];
    FILE* fd;
    char* cp;

    argc--, argv++;
    while (argc > 0 && argv[0][0] == '-') {
        if (strcmp(argv[0], "-v") == 0) {
            if (argc < 1)
                usage();
            argc--, argv++;
            versionFile = argv[0];
        } else if (strcmp(argv[0], "-a") == 0) {
            if (argc < 1)
                usage();
            argc--, argv++;
            alphaFile = argv[0];
        } else
            usage();
        argc--, argv++;
    }
    fd = openFile(versionFile);
    if (fgets(version, sizeof (version)-1, fd) == NULL) {
        fprintf(stderr, "mkversion: No version information in %s.\n",
            versionFile);
        exit(-1);
    }
    cp = strchr(version, '\n');
    if (cp)
        *cp = '\0';
    fclose(fd);
    fd = openFile(alphaFile);
    if (fgets(alpha, sizeof (alpha)-1, fd) == NULL) {
        fprintf(stderr, "mkversion: No alpha information in %s.\n", alphaFile);
        exit(-1);
    }
    fclose(fd);
    cp = strchr(alpha, ' ');            /* skip to 3rd blank-separated field */
```

```
    if (cp)
        cp = strchr(cp+1, ' ');
    if (cp) {                           /* append alpha to version */
        char* tp;
        for (tp = strchr(version, '\0'), cp++; *tp = *cp; tp++, cp++)
            ;
        if (tp[-1] == '\n')
            tp[-1] = '\0';
    } else {
        fprintf(stderr, "mkversion: Malformed alpha information in %s.\n",
            alphaFile);
        exit(-1);
    }
    if (argc > 0) {
        fd = fopen(argv[0], "w");
        if (fd == NULL) {
            fprintf(stderr, "mkversion: %s: Could not open for writing.\n",
                argv[0]);
            exit(-1);
        }
    } else
        fd = stdout;
    fprintf(fd, "#define VERSION \"LIBTIFF, Version %s\\n", version);
    fprintf(fd, "Copyright (c) 1988-1995 Sam Leffler\\n");
    fprintf(fd, "Copyright (c) 1991-1995 Silicon Graphics, Inc.\"\n");

    if (fd != stdout)
        fclose(fd);
    return (0);
}
```

---

kfax'_tiffReadProc() (./kdegraphics/kfax/libtiffax/tif_apple.c:54)

```
_tiffReadProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        return (FSRead((short) fd, (long*) &size, (char*) buf) == noErr ?
            size : (tsize_t) -1);
}

static tsize_t
```

---

kfax'_tiffWriteProc() (./kdegraphics/kfax/libtiffax/tif_apple.c:61)

```
_tiffWriteProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        return (FSWrite((short) fd, (long*) &size, (char*) buf) == noErr ?
            size : (tsize_t) -1);
}

static toff_t
```

---

kfax'_tiffSeekProc() (./kdegraphics/kfax/libtiffax/tif_apple.c:68)

```
_tiffSeekProc(thandle_t fd, toff_t off, int whence)
{
        long fpos, size;

        if (GetEOF((short) fd, &size) != noErr)
                return EOF;
        (void) GetFPos((short) fd, &fpos);

        switch (whence) {
        case SEEK_CUR:
                if (off + fpos > size)
                        SetEOF((short) fd, off + fpos);
                if (SetFPos((short) fd, fsFromMark, off) != noErr)
                        return EOF;
                break;
        case SEEK_END:
                if (off > 0)
                        SetEOF((short) fd, off + size);
                if (SetFPos((short) fd, fsFromStart, off + size) != noErr)
                        return EOF;
                break;
        case SEEK_SET:
                if (off > size)
                        SetEOF((short) fd, off);
                if (SetFPos((short) fd, fsFromStart, off) != noErr)
                        return EOF;
                break;
        }

        return (toff_t)(GetFPos((short) fd, &fpos) == noErr ? fpos : EOF);
}

static int
```

---

kfax'_tiffMapProc() (./kdegraphics/kfax/libtiffax/tif_apple.c:101)

```
_tiffMapProc(thandle_t fd, tdata_t* pbase, toff_t* psize)
{
        return (0);
}

static void
```

---

kfax'_tiffUnmapProc() (./kdegraphics/kfax/libtiffax/tif_apple.c:107)

```
_tiffUnmapProc(thandle_t fd, tdata_t base, toff_t size)
{
}

static int
```

---

kfax'_tiffCloseProc() (./kdegraphics/kfax/libtiffax/tif_apple.c:112)

```
_tiffCloseProc(thandle_t fd)
{
        return (FSClose((short) fd));
}

static toff_t
```

---

kfax'_tiffSizeProc() (./kdegraphics/kfax/libtiffax/tif_apple.c:118)

```
_tiffSizeProc(thandle_t fd)
{
        long size;

        if (GetEOF((short) fd, &size) != noErr) {
                TIFFError("_tiffSizeProc", "%s: Cannot get file size");
                return (-1L);
        }
        return ((toff_t) size);
}

/*
 * Open a TIFF file descriptor for read/writing.
 */
TIFF*
```

---

kfax'TIFFFdOpen() (./kdegraphics/kfax/libtiffax/tif_apple.c:133)

```
TIFFFdOpen(int fd, const char* name, const char* mode)
{
        TIFF* tif;

        tif = TIFFClientOpen(name, mode, (thandle_t) fd,
            _tiffReadProc, _tiffWriteProc, _tiffSeekProc, _tiffCloseProc,
            _tiffSizeProc, _tiffMapProc, _tiffUnmapProc);
        if (tif)
                tif->tif_fd = fd;
        return (tif);
}

/*
 * Open a TIFF file for read/writing.
 */
TIFF*
```

---

kfax'TIFFOpen() (./kdegraphics/kfax/libtiffax/tif_apple.c:149)

```
TIFFOpen(const char* name, const char* mode)
{
        static const char module[] = "TIFFOpen";
        Str255 pname;
        FInfo finfo;
        short fref;
```

```
        OSErr err;

        strcpy((char*) pname, name);
        CtoPstr((char*) pname);

        switch (_TIFFgetMode(mode, module)) {
        default:
                return ((TIFF*) 0);
        case O_RDWR | O_CREAT | O_TRUNC:
                if (GetFInfo(pname, 0, &finfo) == noErr)
                        FSDelete(pname, 0);
                /* fall through */
        case O_RDWR | O_CREAT:
                if ((err = GetFInfo(pname, 0, &finfo)) == fnfErr) {
                        if (Create(pname, 0, '    ', 'TIFF') != noErr)
                                goto badCreate;
                        if (FSOpen(pname, 0, &fref) != noErr)
                                goto badOpen;
                } else if (err == noErr) {
                        if (FSOpen(pname, 0, &fref) != noErr)
                                goto badOpen;
                } else
                        goto badOpen;
                break;
        case O_RDONLY:
        case O_RDWR:
                if (FSOpen(pname, 0, &fref) != noErr)
                        goto badOpen;
                break;
        }
        return (TIFFFdOpen((int) fref, name, mode));
badCreate:
        TIFFError(module, "%s: Cannot create", name);
        return ((TIFF*) 0);
badOpen:
        TIFFError(module, "%s: Cannot open", name);
        return ((TIFF*) 0);
}

void
```

---

kfax'_TIFFmemset() (./kdegraphics/kfax/libtiffax/tif_apple.c:195)

```
_TIFFmemset(tdata_t p, int v, tsize_t c)
{
        memset(p, v, (size_t) c);
}

void
```

---

kfax'_TIFFmemcpy() (./kdegraphics/kfax/libtiffax/tif_apple.c:201)

```
_TIFFmemcpy(tdata_t d, const tdata_t s, tsize_t c)
{
        memcpy(d, s, (size_t) c);
}
```

```
int
```

---

kfax'_TIFFmemcmp() (./kdegraphics/kfax/libtiffax/tif_apple.c:207)

```
_TIFFmemcmp(const tdata_t p1, const tdata_t p2, tsize_t c)
{
        return (memcmp(p1, p2, (size_t) c));
}

tdata_t
```

---

kfax'_TIFFmalloc() (./kdegraphics/kfax/libtiffax/tif_apple.c:213)

```
_TIFFmalloc(tsize_t s)
{
        return (NewPtr((size_t) s));
}

void
```

---

kfax'_TIFFfree() (./kdegraphics/kfax/libtiffax/tif_apple.c:219)

```
_TIFFfree(tdata_t p)
{
        DisposePtr(p);
}

tdata_t
```

---

kfax'_TIFFrealloc() (./kdegraphics/kfax/libtiffax/tif_apple.c:225)

```
_TIFFrealloc(tdata_t p, tsize_t s)
{
        Ptr n = p;

        SetPtrSize(p, (size_t) s);
        if (MemError() && (n = NewPtr((size_t) s)) != NULL) {
                BlockMove(p, n, GetPtrSize(p));
                DisposePtr(p);
        }
        return ((tdata_t) n);
}

static void
```

---

kfax'appleWarningHandler() (./kdegraphics/kfax/libtiffax/tif_apple.c:238)

```
appleWarningHandler(const char* module, const char* fmt, va_list ap)
{
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        fprintf(stderr, "Warning, ");
        vfprintf(stderr, fmt, ap);
        fprintf(stderr, ".\n");
}
```

---

kfax'appleErrorHandler() (./kdegraphics/kfax/libtiffax/tif_apple.c:249)

```
appleErrorHandler(const char* module, const char* fmt, va_list ap)
{
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        vfprintf(stderr, fmt, ap);
        fprintf(stderr, ".\n");
}
```

---

kfax'_tiffReadProc() (./kdegraphics/kfax/libtiffax/tif_atari.c:48)

```
_tiffReadProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        long r;

        r = Fread((int) fd, size, buf);
        if (r < 0) {
                errno = (int)-r;
                r = -1;
        }
        return r;
}

static tsize_t
```

---

kfax'_tiffWriteProc() (./kdegraphics/kfax/libtiffax/tif_atari.c:61)

```
_tiffWriteProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        long r;

        r = Fwrite((int) fd, size, buf);
        if (r < 0) {
                errno = (int)-r;
                r = -1;
        }
        return r;
}

static toff_t
```

---

kfax'_tiffSeekProc() (./kdegraphics/kfax/libtiffax/tif_atari.c:74)

```
_tiffSeekProc(thandle_t fd, off_t off, int whence)
{
        char buf[256];
        long current_off, expected_off, new_off;

        if (whence == SEEK_END || off <= 0)
                return Fseek(off, (int) fd, whence);
        current_off = Fseek(0, (int) fd, SEEK_CUR); /* find out where we are */
        if (whence == SEEK_SET)
                expected_off = off;
        else
                expected_off = off + current_off;
        new_off = Fseek(off, (int) fd, whence);
        if (new_off == expected_off)
                return new_off;
        /* otherwise extend file -- zero filling the hole */
        if (new_off < 0)            /* error? */
                new_off = Fseek(0, (int) fd, SEEK_END); /* go to eof */
        _TIFFmemset(buf, 0, sizeof(buf));
        while (expected_off > new_off) {
                off = expected_off - new_off;
                if (off > sizeof(buf))
                        off = sizeof(buf);
                if ((current_off = Fwrite((int) fd, off, buf)) != off)
                        return (current_off > 0) ?
                            new_off + current_off : new_off;
                new_off += off;
        }
        return new_off;
}

static int
```

---

kfax'_tiffCloseProc() (./kdegraphics/kfax/libtiffax/tif_atari.c:106)

```
_tiffCloseProc(thandle_t fd)
{
        long r;

        r = Fclose((int) fd);
        if (r < 0) {
                errno = (int)-r;
                r = -1;
        }
        return (int)r;
}

static toff_t
```

---

kfax'_tiffSizeProc() (./kdegraphics/kfax/libtiffax/tif_atari.c:119)

```
_tiffSizeProc(thandle_t fd)
```

```
{
        long pos, eof;

        pos = Fseek(0, (int) fd, SEEK_CUR);
        eof = Fseek(0, (int) fd, SEEK_END);
        Fseek(pos, (int) fd, SEEK_SET);
        return eof;
}

static int
```

---

kfax'_tiffMapProc() (./kdegraphics/kfax/libtiffax/tif_atari.c:130)

```
_tiffMapProc(thandle_t fd, tdata_t* pbase, toff_t* psize)
{
        return (0);
}

static void
```

---

kfax'_tiffUnmapProc() (./kdegraphics/kfax/libtiffax/tif_atari.c:136)

```
_tiffUnmapProc(thandle_t fd, tdata_t base, toff_t size)
{
}

/*
* Open a TIFF file descriptor for read/writing.
*/
TIFF*
```

---

kfax'TIFFFdOpen() (./kdegraphics/kfax/libtiffax/tif_atari.c:144)

```
TIFFFdOpen(int fd, const char* name, const char* mode)
{
        TIFF* tif;

        tif = TIFFClientOpen(name, mode,
                (thandle_t) fd,
                _tiffReadProc, _tiffWriteProc, _tiffSeekProc, _tiffCloseProc,
                _tiffSizeProc, _tiffMapProc, _tiffUnmapProc);
        if (tif)
                tif->tif_fd = fd;
        return (tif);
}

/*
* Open a TIFF file for read/writing.
*/
TIFF*
```

---

kfax'TIFFOpen() (./kdegraphics/kfax/libtiffax/tif_atari.c:161)

```
TIFFOpen(const char* name, const char* mode)
{
	static const char module[] = "TIFFOpen";
	int m;
	long fd;

	m = _TIFFgetMode(mode, module);
	if (m == -1)
		return ((TIFF*)0);
	if (m & O_TRUNC) {
		fd = Fcreate(name, 0);
	} else {
		fd = Fopen(name, m & O_ACCMODE);
		if (fd == AEFILNF && m & O_CREAT)
			fd = Fcreate(name, 0);
	}
	if (fd < 0)
		errno = (int)fd;
	if (fd < 0) {
		TIFFError(module, "%s: Cannot open", name);
		return ((TIFF*)0);
	}
	return (TIFFFdOpen(fd, name, mode));
}

#include <stdlib.h>

tdata_t
```

---

kfax'_TIFFmalloc() (./kdegraphics/kfax/libtiffax/tif_atari.c:189)

```
_TIFFmalloc(tsize_t s)
{
	return (malloc((size_t) s));
}

void
```

---

kfax'_TIFFfree() (./kdegraphics/kfax/libtiffax/tif_atari.c:195)

```
_TIFFfree(tdata_t p)
{
	free(p);
}

tdata_t
```

---

kfax'_TIFFrealloc() (./kdegraphics/kfax/libtiffax/tif_atari.c:201)

```
_TIFFrealloc(tdata_t p, tsize_t s)
```

```
{
        return (realloc(p, (size_t) s));
}

void
```

---

kfax'_TIFFmemset() (./kdegraphics/kfax/libtiffax/tif_atari.c:207)

```
_TIFFmemset(tdata_t p, int v, size_t c)
{
        memset(p, v, (size_t) c);
}

void
```

---

kfax'_TIFFmemcpy() (./kdegraphics/kfax/libtiffax/tif_atari.c:213)

```
_TIFFmemcpy(tdata_t d, const tdata_t s, size_t c)
{
        memcpy(d, s, (size_t) c);
}

int
```

---

kfax'_TIFFmemcmp() (./kdegraphics/kfax/libtiffax/tif_atari.c:219)

```
_TIFFmemcmp(const tdata_t p1, const tdata_t p2, tsize_t c)
{
        return (memcmp(p1, p2, (size_t) c));
}

static void
```

---

kfax'atariWarningHandler() (./kdegraphics/kfax/libtiffax/tif_atari.c:225)

```
atariWarningHandler(const char* module, const char* fmt, va_list ap)
{
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        fprintf(stderr, "Warning, ");
        vfprintf(stderr, fmt, ap);
        fprintf(stderr, ".\n");
}
```

---

kfax'atariErrorHandler() (./kdegraphics/kfax/libtiffax/tif_atari.c:236)

```
atariErrorHandler(const char* module, const char* fmt, va_list ap)
{
```

```
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        vfprintf(stderr, fmt, ap);
        fprintf(stderr, ".\n");
}
```

---

kfax'TIFFDefaultTransferFunction() (./kdegraphics/kfax/libtiffax/tif_aux.c:38)

```
TIFFDefaultTransferFunction(TIFFDirectory* td)
{
        uint16 **tf = td->td_transferfunction;
        long i, n = 1<<td->td_bitspersample;

        tf[0] = (uint16 *)_TIFFmalloc(n * sizeof (uint16));
        tf[0][0] = 0;
        for (i = 1; i < n; i++) {
                double t = (double)i/((double) n-1.);
                tf[0][i] = (uint16)floor(65535.*pow(t, 2.2) + .5);
        }
        if (td->td_samplesperpixel - td->td_extrasamples > 1) {
                tf[1] = (uint16 *)_TIFFmalloc(n * sizeof (uint16));
                _TIFFmemcpy(tf[1], tf[0], n * sizeof (uint16));
                tf[2] = (uint16 *)_TIFFmalloc(n * sizeof (uint16));
                _TIFFmemcpy(tf[2], tf[0], n * sizeof (uint16));
        }
}

static void
```

---

kfax'TIFFDefaultRefBlackWhite() (./kdegraphics/kfax/libtiffax/tif_aux.c:58)

```
TIFFDefaultRefBlackWhite(TIFFDirectory* td)
{
        int i;

        td->td_refblackwhite = (float *)_TIFFmalloc(6*sizeof (float));
        for (i = 0; i < 3; i++) {
            td->td_refblackwhite[2*i+0] = 0;
            td->td_refblackwhite[2*i+1] = (float)((1L<<td->td_bitspersample)-1L)
        }
}
#endif

/*
 * Like TIFFGetField, but return any default
 * value if the tag is not present in the directory.
 *
 * NB:  We use the value in the directory, rather than
 *      explcit values so that defaults exist only one
 *      place in the library -- in TIFFDefaultDirectory.
 */
int
```

---

kfax'TIFFVGetFieldDefaulted() (./kdegraphics/kfax/libtiffax/tif_aux.c:79)

```
TIFFVGetFieldDefaulted(TIFF* tif, ttag_t tag, va_list ap)
{
        TIFFDirectory *td = &tif->tif_dir;

        if (TIFFVGetField(tif, tag, ap))
                return (1);
        switch (tag) {
        case TIFFTAG_SUBFILETYPE:
                *va_arg(ap, uint32 *) = td->td_subfiletype;
                return (1);
        case TIFFTAG_BITSPERSAMPLE:
                *va_arg(ap, uint16 *) = td->td_bitspersample;
                return (1);
        case TIFFTAG_THRESHHOLDING:
                *va_arg(ap, uint16 *) = td->td_threshholding;
                return (1);
        case TIFFTAG_FILLORDER:
                *va_arg(ap, uint16 *) = td->td_fillorder;
                return (1);
        case TIFFTAG_ORIENTATION:
                *va_arg(ap, uint16 *) = td->td_orientation;
                return (1);
        case TIFFTAG_SAMPLESPERPIXEL:
                *va_arg(ap, uint16 *) = td->td_samplesperpixel;
                return (1);
        case TIFFTAG_ROWSPERSTRIP:
                *va_arg(ap, uint32 *) = td->td_rowsperstrip;
                return (1);
        case TIFFTAG_MINSAMPLEVALUE:
                *va_arg(ap, uint16 *) = td->td_minsamplevalue;
                return (1);
        case TIFFTAG_MAXSAMPLEVALUE:
                *va_arg(ap, uint16 *) = td->td_maxsamplevalue;
                return (1);
        case TIFFTAG_PLANARCONFIG:
                *va_arg(ap, uint16 *) = td->td_planarconfig;
                return (1);
        case TIFFTAG_RESOLUTIONUNIT:
                *va_arg(ap, uint16 *) = td->td_resolutionunit;
                return (1);
#ifdef CMYK_SUPPORT
        case TIFFTAG_DOTRANGE:
                *va_arg(ap, uint16 *) = 0;
                *va_arg(ap, uint16 *) = (1<<td->td_bitspersample)-1;
                return (1);
        case TIFFTAG_INKSET:
                *va_arg(ap, uint16 *) = td->td_inkset;
                return (1);
#endif
        case TIFFTAG_EXTRASAMPLES:
                *va_arg(ap, uint16 *) = td->td_extrasamples;
                *va_arg(ap, uint16 **) = td->td_sampleinfo;
                return (1);
        case TIFFTAG_MATTEING:
                *va_arg(ap, uint16 *) =
                    (td->td_extrasamples == 1 &&
                     td->td_sampleinfo[0] == EXTRASAMPLE_ASSOCALPHA);
```

```
                return (1);
        case TIFFTAG_TILEDEPTH:
                *va_arg(ap, uint32 *) = td->td_tiledepth;
                return (1);
        case TIFFTAG_DATATYPE:
                *va_arg(ap, uint16 *) = td->td_sampleformat-1;
                return (1);
        case TIFFTAG_IMAGEDEPTH:
                *va_arg(ap, uint32 *) = td->td_imagedepth;
                return (1);
#ifdef YCBCR_SUPPORT
        case TIFFTAG_YCBCRCOEFFICIENTS:
                if (!td->td_ycbcrcoeffs) {
                        td->td_ycbcrcoeffs = (float *)
                            _TIFFmalloc(3*sizeof (float));
                        /* defaults are from CCIR Recommendation 601-1 */
                        td->td_ycbcrcoeffs[0] = 0.299f;
                        td->td_ycbcrcoeffs[1] = 0.587f;
                        td->td_ycbcrcoeffs[2] = 0.114f;
                }
                *va_arg(ap, float **) = td->td_ycbcrcoeffs;
                return (1);
        case TIFFTAG_YCBCRSUBSAMPLING:
                *va_arg(ap, uint16 *) = td->td_ycbcrsubsampling[0];
                *va_arg(ap, uint16 *) = td->td_ycbcrsubsampling[1];
                return (1);
        case TIFFTAG_YCBCRPOSITIONING:
                *va_arg(ap, uint16 *) = td->td_ycbcrpositioning;
                return (1);
#endif
#ifdef COLORIMETRY_SUPPORT
        case TIFFTAG_TRANSFERFUNCTION:
                if (!td->td_transferfunction[0])
                        TIFFDefaultTransferFunction(td);
                *va_arg(ap, uint16 **) = td->td_transferfunction[0];
                if (td->td_samplesperpixel - td->td_extrasamples > 1) {
                        *va_arg(ap, uint16 **) = td->td_transferfunction[1];
                        *va_arg(ap, uint16 **) = td->td_transferfunction[2];
                }
                return (1);
        case TIFFTAG_REFERENCEBLACKWHITE:
                if (!td->td_refblackwhite)
                        TIFFDefaultRefBlackWhite(td);
                *va_arg(ap, float **) = td->td_refblackwhite;
                return (1);
#endif
        }
        return (0);
}

/*
 * Like TIFFGetField, but return any default
 * value if the tag is not present in the directory.
 */
int
```

---

kfax'TIFFGetFieldDefaulted() (./kdegraphics/kfax/libtiffax/tif_aux.c:191)

```
TIFFGetFieldDefaulted(TIFF* tif, ttag_t tag, ...)
{
        int ok;
        va_list ap;

        va_start(ap, tag);
        ok =  TIFFVGetFieldDefaulted(tif, tag, ap);
        va_end(ap);
        return (ok);
}
```

---

kfax'TIFFClose() (./kdegraphics/kfax/libtiffax/tif_close.c:33)

```
TIFFClose(TIFF* tif)
{
        if (tif->tif_mode != O_RDONLY)
                /*
                 * Flush buffered data and directory (if dirty).
                 */
                TIFFFlush(tif);
        if (tif->tif_cleanup)
                (*tif->tif_cleanup)(tif);
        TIFFFreeDirectory(tif);
        if (tif->tif_rawdata && (tif->tif_flags&TIFF_MYBUFFER))
                _TIFFfree(tif->tif_rawdata);
        if (isMapped(tif))
                TIFFUnmapFileContents(tif, tif->tif_base, tif->tif_size);
        (void) TIFFCloseFile(tif);
        if (tif->tif_fieldinfo)
                _TIFFfree(tif->tif_fieldinfo);
        _TIFFfree(tif);
}
```

---

kfax'NotConfigured() (./kdegraphics/kfax/libtiffax/tif_codec.c:92)

```
NotConfigured(TIFF* tif, int scheme)
{
        const TIFFCodec* c = TIFFFindCODEC(scheme);

        TIFFError(tif->tif_name,
            "%s compression support is not configured", c->name);
        return (0);
}
```

---

kfax'TIFFNoEncode() (./kdegraphics/kfax/libtiffax/tif_compress.c:35)

```
TIFFNoEncode(TIFF* tif, char* method)
{
        const TIFFCodec *c = TIFFFindCODEC(tif->tif_dir.td_compression);
        TIFFError(tif->tif_name,
            "%s %s encoding is not implemented", c->name, method);
        return (-1);
```

```
}

int
```

---

kfax'_TIFFNoRowEncode() (./kdegraphics/kfax/libtiffax/tif_compress.c:44)

```
_TIFFNoRowEncode(TIFF* tif, tidata_t pp, tsize_t cc, tsample_t s)
{
        (void) pp; (void) cc; (void) s;
        return (TIFFNoEncode(tif, "scanline"));
}

int
```

---

kfax'_TIFFNoStripEncode() (./kdegraphics/kfax/libtiffax/tif_compress.c:51)

```
_TIFFNoStripEncode(TIFF* tif, tidata_t pp, tsize_t cc, tsample_t s)
{
        (void) pp; (void) cc; (void) s;
        return (TIFFNoEncode(tif, "strip"));
}

int
```

---

kfax'_TIFFNoTileEncode() (./kdegraphics/kfax/libtiffax/tif_compress.c:58)

```
_TIFFNoTileEncode(TIFF* tif, tidata_t pp, tsize_t cc, tsample_t s)
{
        (void) pp; (void) cc; (void) s;
        return (TIFFNoEncode(tif, "tile"));
}

static int
```

---

kfax'TIFFNoDecode() (./kdegraphics/kfax/libtiffax/tif_compress.c:65)

```
TIFFNoDecode(TIFF* tif, char* method)
{
        const TIFFCodec *c = TIFFFindCODEC(tif->tif_dir.td_compression);
        TIFFError(tif->tif_name,
            "%s %s decoding is not implemented", c->name, method);
        return (-1);
}

int
```

---

kfax'_TIFFNoRowDecode() (./kdegraphics/kfax/libtiffax/tif_compress.c:74)

```
_TIFFNoRowDecode(TIFF* tif, tidata_t pp, tsize_t cc, tsample_t s)
{
        (void) pp; (void) cc; (void) s;
        return (TIFFNoDecode(tif, "scanline"));
}

int
```

---

kfax'_TIFFNoStripDecode() (./kdegraphics/kfax/libtiffax/tif_compress.c:81)

```
_TIFFNoStripDecode(TIFF* tif, tidata_t pp, tsize_t cc, tsample_t s)
{
        (void) pp; (void) cc; (void) s;
        return (TIFFNoDecode(tif, "strip"));
}

int
```

---

kfax'_TIFFNoTileDecode() (./kdegraphics/kfax/libtiffax/tif_compress.c:88)

```
_TIFFNoTileDecode(TIFF* tif, tidata_t pp, tsize_t cc, tsample_t s)
{
        (void) pp; (void) cc; (void) s;
        return (TIFFNoDecode(tif, "tile"));
}

int
```

---

kfax'_TIFFNoSeek() (./kdegraphics/kfax/libtiffax/tif_compress.c:95)

```
_TIFFNoSeek(TIFF* tif, uint32 off)
{
        (void) off;
        TIFFError(tif->tif_name,
            "Compression algorithm does not support random access");
        return (0);
}

int
```

---

kfax'_TIFFNoPreCode() (./kdegraphics/kfax/libtiffax/tif_compress.c:104)

```
_TIFFNoPreCode(TIFF* tif, tsample_t s)
{
        (void) tif; (void) s;
        return (1);
}
```

---

kfax'_TIFFtrue() (./kdegraphics/kfax/libtiffax/tif_compress.c:110)

```
static int _TIFFtrue(TIFF* tif) { (void) tif; return (1); }
```

---

kfax'_TIFFvoid() (./kdegraphics/kfax/libtiffax/tif_compress.c:111)

```
static void _TIFFvoid(TIFF* tif) { (void) tif; }

int
```

---

kfax'TIFFSetCompressionScheme() (./kdegraphics/kfax/libtiffax/tif_compress.c:114)

```
TIFFSetCompressionScheme(TIFF* tif, int scheme)
{
        const TIFFCodec *c = TIFFFindCODEC(scheme);

        if (!c) {
                TIFFError(tif->tif_name,
                    "Unknown data compression algorithm %u (0x%x)",
                    scheme, scheme);
                return (0);
        }
        tif->tif_setupdecode = _TIFFtrue;
        tif->tif_predecode = _TIFFNoPreCode;
        tif->tif_decoderow = _TIFFNoRowDecode;
        tif->tif_decodestrip = _TIFFNoStripDecode;
        tif->tif_decodetile = _TIFFNoTileDecode;
        tif->tif_setupencode = _TIFFtrue;
        tif->tif_preencode = _TIFFNoPreCode;
        tif->tif_postencode = _TIFFtrue;
        tif->tif_encoderow = _TIFFNoRowEncode;
        tif->tif_encodestrip = _TIFFNoStripEncode;
        tif->tif_encodetile = _TIFFNoTileEncode;
        tif->tif_close = _TIFFvoid;
        tif->tif_seek = _TIFFNoSeek;
        tif->tif_cleanup = _TIFFvoid;
        tif->tif_defstripsize = _TIFFDefaultStripSize;
        tif->tif_deftilesize = _TIFFDefaultTileSize;
        tif->tif_flags &= ~TIFF_NOBITREV;
        return ((*c->init)(tif, scheme));
}

/*
 * Other compression schemes may be registered.  Registered
 * schemes can also override the builtin versions provided
 * by this library.
 */
```

---

kfax'TIFFFindCODEC() (./kdegraphics/kfax/libtiffax/tif_compress.c:156)

```
TIFFFindCODEC(uint16 scheme)
{
        const TIFFCodec* c;
        codec_t* cd;

        for (cd = registeredCODECS; cd; cd = cd->next)
                if (cd->info->scheme == scheme)
                        return ((const TIFFCodec*) cd->info);
        for (c = _TIFFBuiltinCODECS; c->name; c++)
                if (c->scheme == scheme)
                        return (c);
        return ((const TIFFCodec*) 0);
}

TIFFCodec*
```

---

kfax'TIFFRegisterCODEC() (./kdegraphics/kfax/libtiffax/tif_compress.c:171)

```
TIFFRegisterCODEC(uint16 scheme, const char* name, TIFFInitMethod init)
{
        codec_t* cd = (codec_t*)
            _TIFFmalloc(sizeof (codec_t) + sizeof (TIFFCodec) + strlen(name)+1);

        if (cd != NULL) {
                cd->info = (TIFFCodec*) ((tidata_t) cd + sizeof (codec_t));
                cd->info->name = (char*)
                    ((tidata_t) cd->info + sizeof (TIFFCodec));
                strcpy(cd->info->name, name);
                cd->info->scheme = scheme;
                cd->info->init = init;
                cd->next = registeredCODECS;
                registeredCODECS = cd;
        } else
                TIFFError("TIFFRegisterCODEC",
                    "No space to register compression scheme %s", name);
        return (cd->info);
}

void
```

---

kfax'TIFFUnRegisterCODEC() (./kdegraphics/kfax/libtiffax/tif_compress.c:192)

```
TIFFUnRegisterCODEC(TIFFCodec* c)
{
        codec_t* cd;
        codec_t** pcd;

        for (pcd = ®isteredCODECS; (cd = *pcd); pcd = &cd->next)
                if (cd->info == c) {
                        *pcd = cd->next;
                        _TIFFfree(cd);
                        return;
                }
```

```
        TIFFError("TIFFUnRegisterCODEC",
            "Cannot remove compression scheme %s; not registered", c->name);
}
```

kfax'_TIFFsetByteArray() (./kdegraphics/kfax/libtiffax/tif_dir.c:44)

```
_TIFFsetByteArray(void** vpp, void* vp, long n)
{
        if (*vpp)
                _TIFFfree(*vpp), *vpp = 0;
        if (vp && (*vpp = (void*) _TIFFmalloc(n)))
                _TIFFmemcpy(*vpp, vp, n);
}
```

kfax'_TIFFsetString() (./kdegraphics/kfax/libtiffax/tif_dir.c:51)

```
void _TIFFsetString(char** cpp, char* cp)
    { _TIFFsetByteArray((void**) cpp, (void*) cp, (long) (strlen(cp)+1)); }
```

kfax'_TIFFsetShortArray() (./kdegraphics/kfax/libtiffax/tif_dir.c:53)

```
void _TIFFsetShortArray(uint16** wpp, uint16* wp, long n)
    { _TIFFsetByteArray((void**) wpp, (void*) wp, n*sizeof (uint16)); }
```

kfax'_TIFFsetLongArray() (./kdegraphics/kfax/libtiffax/tif_dir.c:55)

```
void _TIFFsetLongArray(uint32** lpp, uint32* lp, long n)
    { _TIFFsetByteArray((void**) lpp, (void*) lp, n*sizeof (uint32)); }
```

kfax'_TIFFsetFloatArray() (./kdegraphics/kfax/libtiffax/tif_dir.c:57)

```
void _TIFFsetFloatArray(float** fpp, float* fp, long n)
    { _TIFFsetByteArray((void**) fpp, (void*) fp, n*sizeof (float)); }
```

kfax'_TIFFsetDoubleArray() (./kdegraphics/kfax/libtiffax/tif_dir.c:59)

```
void _TIFFsetDoubleArray(double** dpp, double* dp, long n)
    { _TIFFsetByteArray((void**) dpp, (void*) dp, n*sizeof (double)); }

/*
 * Install extra samples information.
 */
static int
```

kfax'setExtraSamples() (./kdegraphics/kfax/libtiffax/tif_dir.c:66)

```
setExtraSamples(TIFFDirectory* td, va_list ap, int* v)
{
        uint16* va;
        int i;

        *v = va_arg(ap, int);
        if ((uint16) *v > td->td_samplesperpixel)
                return (0);
        va = va_arg(ap, uint16*);
        if (*v > 0 && va == NULL)               /* typically missing param */
                return (0);
        for (i = 0; i < *v; i++)
                if (va[i] > EXTRASAMPLE_UNASSALPHA)
                        return (0);
        td->td_extrasamples = (uint16) *v;
        _TIFFsetShortArray(&td->td_sampleinfo, va, td->td_extrasamples);
        return (1);
}

static int
```

---

kfax'_TIFFVSetField() (./kdegraphics/kfax/libtiffax/tif_dir.c:86)

```
_TIFFVSetField(TIFF* tif, ttag_t tag, va_list ap)
{
        TIFFDirectory* td = &tif->tif_dir;
        int status = 1;
        uint32 v32;
        int i, v;

        switch (tag) {
        case TIFFTAG_SUBFILETYPE:
                td->td_subfiletype = va_arg(ap, uint32);
                break;
        case TIFFTAG_IMAGEWIDTH:
                td->td_imagewidth = va_arg(ap, uint32);
                break;
        case TIFFTAG_IMAGELENGTH:
                td->td_imagelength = va_arg(ap, uint32);
                break;
        case TIFFTAG_BITSPERSAMPLE:
                td->td_bitspersample = (uint16) va_arg(ap, int);
                /*
                 * If the data require post-decoding processing
                 * to byte-swap samples, set it up here.  Note
                 * that since tags are required to be ordered,
                 * compression code can override this behaviour
                 * in the setup method if it wants to roll the
                 * post decoding work in with its normal work.
                 */
                if (tif->tif_flags & TIFF_SWAB) {
                        if (td->td_bitspersample == 16)
                                tif->tif_postdecode = _TIFFSwab16BitData;
                        else if (td->td_bitspersample == 32)
                                tif->tif_postdecode = _TIFFSwab32BitData;
```

```
                else if (td->td_bitspersample == 64)
                        tif->tif_postdecode = _TIFFSwab64BitData;
        }
        break;
case TIFFTAG_COMPRESSION:
        v = va_arg(ap, int) & 0xffff;
        /*
         * If we're changing the compression scheme,
         * the notify the previous module so that it
         * can cleanup any state it's setup.
         */
        if (TIFFFieldSet(tif, FIELD_COMPRESSION)) {
                if (td->td_compression == v)
                        break;
                (*tif->tif_cleanup)(tif);
                tif->tif_flags &= ~TIFF_CODERSETUP;
        }
        /*
         * Setup new compression routine state.
         */
        if (status = TIFFSetCompressionScheme(tif, v))
                td->td_compression = v;
        break;
case TIFFTAG_PHOTOMETRIC:
        td->td_photometric = (uint16) va_arg(ap, int);
        break;
case TIFFTAG_THRESHHOLDING:
        td->td_threshholding = (uint16) va_arg(ap, int);
        break;
case TIFFTAG_FILLORDER:
        v = va_arg(ap, int);
        if (v != FILLORDER_LSB2MSB && v != FILLORDER_MSB2LSB)
                goto badvalue;
        td->td_fillorder = (uint16) v;
        break;
case TIFFTAG_DOCUMENTNAME:
        _TIFFsetString(&td->td_documentname, va_arg(ap, char*));
        break;
case TIFFTAG_ARTIST:
        _TIFFsetString(&td->td_artist, va_arg(ap, char*));
        break;
case TIFFTAG_DATETIME:
        _TIFFsetString(&td->td_datetime, va_arg(ap, char*));
        break;
case TIFFTAG_HOSTCOMPUTER:
        _TIFFsetString(&td->td_hostcomputer, va_arg(ap, char*));
        break;
case TIFFTAG_IMAGEDESCRIPTION:
        _TIFFsetString(&td->td_imagedescription, va_arg(ap, char*));
        break;
case TIFFTAG_MAKE:
        _TIFFsetString(&td->td_make, va_arg(ap, char*));
        break;
case TIFFTAG_MODEL:
        _TIFFsetString(&td->td_model, va_arg(ap, char*));
        break;
case TIFFTAG_SOFTWARE:
        _TIFFsetString(&td->td_software, va_arg(ap, char*));
        break;
case TIFFTAG_ORIENTATION:
        v = va_arg(ap, int);
```

```
		if (v < ORIENTATION_TOPLEFT || ORIENTATION_LEFTBOT < v) {
			TIFFWarning(tif->tif_name,
			    "Bad value %ld for \"%s\" tag ignored",
			    v, _TIFFFieldWithTag(tif, tag)->field_name);
		} else
			td->td_orientation = (uint16) v;
		break;
	case TIFFTAG_SAMPLESPERPIXEL:
		/* XXX should cross check -- e.g. if pallette, then 1 */
		v = va_arg(ap, int);
		if (v == 0)
			goto badvalue;
		td->td_samplesperpixel = (uint16) v;
		break;
	case TIFFTAG_ROWSPERSTRIP:
		v32 = va_arg(ap, uint32);
		if (v32 == 0)
			goto badvalue32;
		td->td_rowsperstrip = v32;
		if (!TIFFFieldSet(tif, FIELD_TILEDIMENSIONS)) {
			td->td_tilelength = v32;
			td->td_tilewidth = td->td_imagewidth;
		}
		break;
	case TIFFTAG_MINSAMPLEVALUE:
		td->td_minsamplevalue = (uint16) va_arg(ap, int);
		break;
	case TIFFTAG_MAXSAMPLEVALUE:
		td->td_maxsamplevalue = (uint16) va_arg(ap, int);
		break;
	case TIFFTAG_SMINSAMPLEVALUE:
		td->td_sminsamplevalue = (double) va_arg(ap, dblparam_t);
		break;
	case TIFFTAG_SMAXSAMPLEVALUE:
		td->td_smaxsamplevalue = (double) va_arg(ap, dblparam_t);
		break;
	case TIFFTAG_XRESOLUTION:
		td->td_xresolution = (float) va_arg(ap, dblparam_t);
		break;
	case TIFFTAG_YRESOLUTION:
		td->td_yresolution = (float) va_arg(ap, dblparam_t);
		break;
	case TIFFTAG_PLANARCONFIG:
		v = va_arg(ap, int);
		if (v != PLANARCONFIG_CONTIG && v != PLANARCONFIG_SEPARATE)
			goto badvalue;
		td->td_planarconfig = (uint16) v;
		break;
	case TIFFTAG_PAGENAME:
		_TIFFsetString(&td->td_pagename, va_arg(ap, char*));
		break;
	case TIFFTAG_XPOSITION:
		td->td_xposition = (float) va_arg(ap, dblparam_t);
		break;
	case TIFFTAG_YPOSITION:
		td->td_yposition = (float) va_arg(ap, dblparam_t);
		break;
	case TIFFTAG_RESOLUTIONUNIT:
		v = va_arg(ap, int);
		if (v < RESUNIT_NONE || RESUNIT_CENTIMETER < v)
			goto badvalue;
```

```
                td->td_resolutionunit = (uint16) v;
                break;
        case TIFFTAG_PAGENUMBER:
                td->td_pagenumber[0] = (uint16) va_arg(ap, int);
                td->td_pagenumber[1] = (uint16) va_arg(ap, int);
                break;
        case TIFFTAG_HALFTONEHINTS:
                td->td_halftonehints[0] = (uint16) va_arg(ap, int);
                td->td_halftonehints[1] = (uint16) va_arg(ap, int);
                break;
        case TIFFTAG_COLORMAP:
                v32 = (uint32)(1L<<td->td_bitspersample);
                _TIFFsetShortArray(&td->td_colormap[0], va_arg(ap, uint16*), v32
                _TIFFsetShortArray(&td->td_colormap[1], va_arg(ap, uint16*), v32
                _TIFFsetShortArray(&td->td_colormap[2], va_arg(ap, uint16*), v32
                break;
        case TIFFTAG_EXTRASAMPLES:
                if (!setExtraSamples(td, ap, &v))
                        goto badvalue;
                break;
        case TIFFTAG_MATTEING:
                td->td_extrasamples = (uint16) (va_arg(ap, int) != 0);
                if (td->td_extrasamples) {
                        uint16 sv = EXTRASAMPLE_ASSOCALPHA;
                        _TIFFsetShortArray(&td->td_sampleinfo, &sv, 1);
                }
                break;
        case TIFFTAG_TILEWIDTH:
                v32 = va_arg(ap, uint32);
                if (v32 % 16) {
                        if (tif->tif_mode != O_RDONLY)
                                goto badvalue32;
                        TIFFWarning(tif->tif_name,
                            "Nonstandard tile width %d, convert file", v32);
                }
                td->td_tilewidth = v32;
                tif->tif_flags |= TIFF_ISTILED;
                break;
        case TIFFTAG_TILELENGTH:
                v32 = va_arg(ap, uint32);
                if (v32 % 16) {
                        if (tif->tif_mode != O_RDONLY)
                                goto badvalue32;
                        TIFFWarning(tif->tif_name,
                            "Nonstandard tile length %d, convert file", v32);
                }
                td->td_tilelength = v32;
                tif->tif_flags |= TIFF_ISTILED;
                break;
        case TIFFTAG_TILEDEPTH:
                v32 = va_arg(ap, uint32);
                if (v32 == 0)
                        goto badvalue32;
                td->td_tiledepth = v32;
                break;
        case TIFFTAG_DATATYPE:
                v = va_arg(ap, int);
                switch (v) {
                case DATATYPE_VOID:     v = SAMPLEFORMAT_VOID;  break;
                case DATATYPE_INT:      v = SAMPLEFORMAT_INT;   break;
                case DATATYPE_UINT:     v = SAMPLEFORMAT_UINT;  break;
```

```
                case DATATYPE_IEEEFP:   v = SAMPLEFORMAT_IEEEFP;break;
                default:                goto badvalue;
                }
                td->td_sampleformat = (uint16) v;
                break;
        case TIFFTAG_SAMPLEFORMAT:
                v = va_arg(ap, int);
                if (v < SAMPLEFORMAT_UINT || SAMPLEFORMAT_VOID < v)
                        goto badvalue;
                td->td_sampleformat = (uint16) v;
                break;
        case TIFFTAG_IMAGEDEPTH:
                td->td_imagedepth = va_arg(ap, uint32);
                break;
#if SUBIFD_SUPPORT
        case TIFFTAG_SUBIFD:
                if ((tif->tif_flags & TIFF_INSUBIFD) == 0) {
                        td->td_nsubifd = (uint16) va_arg(ap, int);
                        _TIFFsetLongArray(&td->td_subifd, va_arg(ap, uint32*),
                            (long) td->td_nsubifd);
                } else {
                        TIFFError(tif->tif_name, "Sorry, cannot nest SubIFDs");
                        status = 0;
                }
                break;
#endif
#ifdef YCBCR_SUPPORT
        case TIFFTAG_YCBCRCOEFFICIENTS:
                _TIFFsetFloatArray(&td->td_ycbcrcoeffs, va_arg(ap, float*), 3);
                break;
        case TIFFTAG_YCBCRPOSITIONING:
                td->td_ycbcrpositioning = (uint16) va_arg(ap, int);
                break;
        case TIFFTAG_YCBCRSUBSAMPLING:
                td->td_ycbcrsubsampling[0] = (uint16) va_arg(ap, int);
                td->td_ycbcrsubsampling[1] = (uint16) va_arg(ap, int);
                break;
#endif
#ifdef COLORIMETRY_SUPPORT
        case TIFFTAG_WHITEPOINT:
                _TIFFsetFloatArray(&td->td_whitepoint, va_arg(ap, float*), 2);
                break;
        case TIFFTAG_PRIMARYCHROMATICITIES:
                _TIFFsetFloatArray(&td->td_primarychromas, va_arg(ap, float*), 6
                break;
        case TIFFTAG_TRANSFERFUNCTION:
                v = (td->td_samplesperpixel - td->td_extrasamples) > 1 ? 3 : 1;
                for (i = 0; i < v; i++)
                        _TIFFsetShortArray(&td->td_transferfunction[i],
                            va_arg(ap, uint16*), 1L<<td->td_bitspersample);
                break;
        case TIFFTAG_REFERENCEBLACKWHITE:
                /* XXX should check for null range */
                _TIFFsetFloatArray(&td->td_refblackwhite, va_arg(ap, float*), 6)
                break;
#endif
#ifdef CMYK_SUPPORT
        case TIFFTAG_INKSET:
                td->td_inkset = (uint16) va_arg(ap, int);
                break;
        case TIFFTAG_DOTRANGE:
```

```
                /* XXX should check for null range */
                td->td_dotrange[0] = (uint16) va_arg(ap, int);
                td->td_dotrange[1] = (uint16) va_arg(ap, int);
                break;
        case TIFFTAG_INKNAMES:
                _TIFFsetString(&td->td_inknames, va_arg(ap, char*));
                break;
        case TIFFTAG_TARGETPRINTER:
                _TIFFsetString(&td->td_targetprinter, va_arg(ap, char*));
                break;
#endif
        default:
                TIFFError(tif->tif_name,
                    "Internal error, tag value botch, tag \"%s\"",
                    _TIFFFieldWithTag(tif, tag)->field_name);
                status = 0;
                break;
        }
        if (status) {
                TIFFSetFieldBit(tif, _TIFFFieldWithTag(tif, tag)->field_bit);
                tif->tif_flags |= TIFF_DIRTYDIRECT;
        }
        va_end(ap);
        return (status);
badvalue:
        TIFFError(tif->tif_name, "%d: Bad value for \"%s\"", v,
            _TIFFFieldWithTag(tif, tag)->field_name);
        va_end(ap);
        return (0);
badvalue32:
        TIFFError(tif->tif_name, "%ld: Bad value for \"%s\"", v32,
            _TIFFFieldWithTag(tif, tag)->field_name);
        va_end(ap);
        return (0);
}

/*
 * Return 1/0 according to whether or not
 * it is permissible to set the tag's value.
 * Note that we allow ImageLength to be changed
 * so that we can append and extend to images.
 * Any other tag may not be altered once writing
 * has commenced, unless its value has no effect
 * on the format of the data that is written.
 */
static int
```

---

kfax'OkToChangeTag() (./kdegraphics/kfax/libtiffax/tif_dir.c:408)

```
OkToChangeTag(TIFF* tif, ttag_t tag)
{
        if (tag != TIFFTAG_IMAGELENGTH &&
            (tif->tif_flags & TIFF_BEENWRITING)) {
                const TIFFFieldInfo *fip = _TIFFFindFieldInfo(tif, tag, TIFF_ANY
                /*
                 * Consult info table to see if tag can be changed
                 * after we've started writing.  We only allow changes
                 * to those tags that don't/shouldn't affect the
```

```
                 * compression and/or format of the data.
                 */
                if (fip && !fip->field_oktochange) {
                        TIFFError("TIFFSetField",
                            "%s: Cannot modify tag \"%s\" while writing",
                            tif->tif_name, fip->field_name);
                        return (0);
                }
        }
        return (1);
}

/*
 * Record the value of a field in the
 * internal directory structure.  The
 * field will be written to the file
 * when/if the directory structure is
 * updated.
 */
int
```

---

kfax'TIFFSetField() (./kdegraphics/kfax/libtiffax/tif_dir.c:437)

```
TIFFSetField(TIFF* tif, ttag_t tag, ...)
{
        va_list ap;
        int status;

        va_start(ap, tag);
        status = TIFFVSetField(tif, tag, ap);
        va_end(ap);
        return (status);
}

/*
 * Like TIFFSetField, but taking a varargs
 * parameter list.  This routine is useful
 * for building higher-level interfaces on
 * top of the library.
 */
int
```

---

kfax'TIFFVSetField() (./kdegraphics/kfax/libtiffax/tif_dir.c:455)

```
TIFFVSetField(TIFF* tif, ttag_t tag, va_list ap)
{
        return OkToChangeTag(tif, tag) ?
            (*tif->tif_vsetfield)(tif, tag, ap) : 0;
}

static int
```

---

kfax'_TIFFVGetField() (./kdegraphics/kfax/libtiffax/tif_dir.c:462)

```
_TIFFVGetField(TIFF* tif, ttag_t tag, va_list ap)
{
        TIFFDirectory* td = &tif->tif_dir;

        switch (tag) {
        case TIFFTAG_SUBFILETYPE:
                *va_arg(ap, uint32*) = td->td_subfiletype;
                break;
        case TIFFTAG_IMAGEWIDTH:
                *va_arg(ap, uint32*) = td->td_imagewidth;
                break;
        case TIFFTAG_IMAGELENGTH:
                *va_arg(ap, uint32*) = td->td_imagelength;
                break;
        case TIFFTAG_BITSPERSAMPLE:
                *va_arg(ap, uint16*) = td->td_bitspersample;
                break;
        case TIFFTAG_COMPRESSION:
                *va_arg(ap, uint16*) = td->td_compression;
                break;
        case TIFFTAG_PHOTOMETRIC:
                *va_arg(ap, uint16*) = td->td_photometric;
                break;
        case TIFFTAG_THRESHHOLDING:
                *va_arg(ap, uint16*) = td->td_threshholding;
                break;
        case TIFFTAG_FILLORDER:
                *va_arg(ap, uint16*) = td->td_fillorder;
                break;
        case TIFFTAG_DOCUMENTNAME:
                *va_arg(ap, char**) = td->td_documentname;
                break;
        case TIFFTAG_ARTIST:
                *va_arg(ap, char**) = td->td_artist;
                break;
        case TIFFTAG_DATETIME:
                *va_arg(ap, char**) = td->td_datetime;
                break;
        case TIFFTAG_HOSTCOMPUTER:
                *va_arg(ap, char**) = td->td_hostcomputer;
                break;
        case TIFFTAG_IMAGEDESCRIPTION:
                *va_arg(ap, char**) = td->td_imagedescription;
                break;
        case TIFFTAG_MAKE:
                *va_arg(ap, char**) = td->td_make;
                break;
        case TIFFTAG_MODEL:
                *va_arg(ap, char**) = td->td_model;
                break;
        case TIFFTAG_SOFTWARE:
                *va_arg(ap, char**) = td->td_software;
                break;
        case TIFFTAG_ORIENTATION:
                *va_arg(ap, uint16*) = td->td_orientation;
                break;
        case TIFFTAG_SAMPLESPERPIXEL:
                *va_arg(ap, uint16*) = td->td_samplesperpixel;
                break;
        case TIFFTAG_ROWSPERSTRIP:
```

```
                *va_arg(ap, uint32*) = td->td_rowsperstrip;
                break;
        case TIFFTAG_MINSAMPLEVALUE:
                *va_arg(ap, uint16*) = td->td_minsamplevalue;
                break;
        case TIFFTAG_MAXSAMPLEVALUE:
                *va_arg(ap, uint16*) = td->td_maxsamplevalue;
                break;
        case TIFFTAG_SMINSAMPLEVALUE:
                *va_arg(ap, double*) = td->td_sminsamplevalue;
                break;
        case TIFFTAG_SMAXSAMPLEVALUE:
                *va_arg(ap, double*) = td->td_smaxsamplevalue;
                break;
        case TIFFTAG_XRESOLUTION:
                *va_arg(ap, float*) = td->td_xresolution;
                break;
        case TIFFTAG_YRESOLUTION:
                *va_arg(ap, float*) = td->td_yresolution;
                break;
        case TIFFTAG_PLANARCONFIG:
                *va_arg(ap, uint16*) = td->td_planarconfig;
                break;
        case TIFFTAG_XPOSITION:
                *va_arg(ap, float*) = td->td_xposition;
                break;
        case TIFFTAG_YPOSITION:
                *va_arg(ap, float*) = td->td_yposition;
                break;
        case TIFFTAG_PAGENAME:
                *va_arg(ap, char**) = td->td_pagename;
                break;
        case TIFFTAG_RESOLUTIONUNIT:
                *va_arg(ap, uint16*) = td->td_resolutionunit;
                break;
        case TIFFTAG_PAGENUMBER:
                *va_arg(ap, uint16*) = td->td_pagenumber[0];
                *va_arg(ap, uint16*) = td->td_pagenumber[1];
                break;
        case TIFFTAG_HALFTONEHINTS:
                *va_arg(ap, uint16*) = td->td_halftonehints[0];
                *va_arg(ap, uint16*) = td->td_halftonehints[1];
                break;
        case TIFFTAG_COLORMAP:
                *va_arg(ap, uint16**) = td->td_colormap[0];
                *va_arg(ap, uint16**) = td->td_colormap[1];
                *va_arg(ap, uint16**) = td->td_colormap[2];
                break;
        case TIFFTAG_STRIPOFFSETS:
        case TIFFTAG_TILEOFFSETS:
                *va_arg(ap, uint32**) = td->td_stripoffset;
                break;
        case TIFFTAG_STRIPBYTECOUNTS:
        case TIFFTAG_TILEBYTECOUNTS:
                *va_arg(ap, uint32**) = td->td_stripbytecount;
                break;
        case TIFFTAG_MATTEING:
                *va_arg(ap, uint16*) =
                    (td->td_extrasamples == 1 &&
                     td->td_sampleinfo[0] == EXTRASAMPLE_ASSOCALPHA);
                break;
```

```
	case TIFFTAG_EXTRASAMPLES:
		*va_arg(ap, uint16*) = td->td_extrasamples;
		*va_arg(ap, uint16**) = td->td_sampleinfo;
		break;
	case TIFFTAG_TILEWIDTH:
		*va_arg(ap, uint32*) = td->td_tilewidth;
		break;
	case TIFFTAG_TILELENGTH:
		*va_arg(ap, uint32*) = td->td_tilelength;
		break;
	case TIFFTAG_TILEDEPTH:
		*va_arg(ap, uint32*) = td->td_tiledepth;
		break;
	case TIFFTAG_DATATYPE:
		switch (td->td_sampleformat) {
		case SAMPLEFORMAT_UINT:
			*va_arg(ap, uint16*) = DATATYPE_UINT;
			break;
		case SAMPLEFORMAT_INT:
			*va_arg(ap, uint16*) = DATATYPE_INT;
			break;
		case SAMPLEFORMAT_IEEEFP:
			*va_arg(ap, uint16*) = DATATYPE_IEEEFP;
			break;
		case SAMPLEFORMAT_VOID:
			*va_arg(ap, uint16*) = DATATYPE_VOID;
			break;
		}
		break;
	case TIFFTAG_SAMPLEFORMAT:
		*va_arg(ap, uint16*) = td->td_sampleformat;
		break;
	case TIFFTAG_IMAGEDEPTH:
		*va_arg(ap, uint32*) = td->td_imagedepth;
		break;
#if SUBIFD_SUPPORT
	case TIFFTAG_SUBIFD:
		*va_arg(ap, uint16*) = td->td_nsubifd;
		*va_arg(ap, uint32**) = td->td_subifd;
		break;
#endif
#ifdef YCBCR_SUPPORT
	case TIFFTAG_YCBCRCOEFFICIENTS:
		*va_arg(ap, float**) = td->td_ycbcrcoeffs;
		break;
	case TIFFTAG_YCBCRPOSITIONING:
		*va_arg(ap, uint16*) = td->td_ycbcrpositioning;
		break;
	case TIFFTAG_YCBCRSUBSAMPLING:
		*va_arg(ap, uint16*) = td->td_ycbcrsubsampling[0];
		*va_arg(ap, uint16*) = td->td_ycbcrsubsampling[1];
		break;
#endif
#ifdef COLORIMETRY_SUPPORT
	case TIFFTAG_WHITEPOINT:
		*va_arg(ap, float**) = td->td_whitepoint;
		break;
	case TIFFTAG_PRIMARYCHROMATICITIES:
		*va_arg(ap, float**) = td->td_primarychromas;
		break;
	case TIFFTAG_TRANSFERFUNCTION:
```

```
                *va_arg(ap, uint16**) = td->td_transferfunction[0];
                if (td->td_samplesperpixel - td->td_extrasamples > 1) {
                        *va_arg(ap, uint16**) = td->td_transferfunction[1];
                        *va_arg(ap, uint16**) = td->td_transferfunction[2];
                }
                break;
        case TIFFTAG_REFERENCEBLACKWHITE:
                *va_arg(ap, float**) = td->td_refblackwhite;
                break;
#endif
#ifdef CMYK_SUPPORT
        case TIFFTAG_INKSET:
                *va_arg(ap, uint16*) = td->td_inkset;
                break;
        case TIFFTAG_DOTRANGE:
                *va_arg(ap, uint16*) = td->td_dotrange[0];
                *va_arg(ap, uint16*) = td->td_dotrange[1];
                break;
        case TIFFTAG_INKNAMES:
                *va_arg(ap, char**) = td->td_inknames;
                break;
        case TIFFTAG_TARGETPRINTER:
                *va_arg(ap, char**) = td->td_targetprinter;
                break;
#endif
        default:
                TIFFError("_TIFFVGetField",
                    "Internal error, no value returned for tag \"%s\"",
                    _TIFFFieldWithTag(tif, tag)->field_name);
                break;
        }
        return (1);
}

/*
 * Return the value of a field in the
 * internal directory structure.
 */
int
```

---

kfax'TIFFGetField() (./kdegraphics/kfax/libtiffax/tif_dir.c:683)

```
TIFFGetField(TIFF* tif, ttag_t tag, ...)
{
        int status;
        va_list ap;

        va_start(ap, tag);
        status = TIFFVGetField(tif, tag, ap);
        va_end(ap);
        return (status);
}

/*
 * Like TIFFGetField, but taking a varargs
 * parameter list.  This routine is useful
 * for building higher-level interfaces on
 * top of the library.
```

```
 */
int
```

---

kfax'TIFFVGetField() (./kdegraphics/kfax/libtiffax/tif_dir.c:701)

```
TIFFVGetField(TIFF* tif, ttag_t tag, va_list ap)
{
        const TIFFFieldInfo* fip = _TIFFFindFieldInfo(tif, tag, TIFF_ANY);
        return (fip && TIFFFieldSet(tif, fip->field_bit) ?
            (*tif->tif_vgetfield)(tif, tag, ap) : 0);
}
```

---

kfax'TIFFFreeDirectory() (./kdegraphics/kfax/libtiffax/tif_dir.c:719)

```
TIFFFreeDirectory(TIFF* tif)
{
        register TIFFDirectory *td = &tif->tif_dir;

        CleanupField(td_colormap[0]);
        CleanupField(td_colormap[1]);
        CleanupField(td_colormap[2]);
        CleanupField(td_documentname);
        CleanupField(td_artist);
        CleanupField(td_datetime);
        CleanupField(td_hostcomputer);
        CleanupField(td_imagedescription);
        CleanupField(td_make);
        CleanupField(td_model);
        CleanupField(td_software);
        CleanupField(td_pagename);
        CleanupField(td_sampleinfo);
#if SUBIFD_SUPPORT
        CleanupField(td_subifd);
#endif
#ifdef YCBCR_SUPPORT
        CleanupField(td_ycbcrcoeffs);
#endif
#ifdef CMYK_SUPPORT
        CleanupField(td_inknames);
        CleanupField(td_targetprinter);
#endif
#ifdef COLORIMETRY_SUPPORT
        CleanupField(td_whitepoint);
        CleanupField(td_primarychromas);
        CleanupField(td_refblackwhite);
        CleanupField(td_transferfunction[0]);
        CleanupField(td_transferfunction[1]);
        CleanupField(td_transferfunction[2]);
#endif
        CleanupField(td_stripoffset);
        CleanupField(td_stripbytecount);
}
```

---

kfax'TIFFSetTagExtender() (./kdegraphics/kfax/libtiffax/tif_dir.c:765)

```
TIFFSetTagExtender(TIFFExtendProc extender)
{
        TIFFExtendProc prev = _TIFFextender;
        _TIFFextender = extender;
        return (prev);
}

/*
 * Setup a default directory structure.
 */
int
```

---

kfax'TIFFDefaultDirectory() (./kdegraphics/kfax/libtiffax/tif_dir.c:776)

```
TIFFDefaultDirectory(TIFF* tif)
{
        register TIFFDirectory* td = &tif->tif_dir;

        _TIFFSetupFieldInfo(tif);
        _TIFFmemset(td, 0, sizeof (*td));
        td->td_fillorder = FILLORDER_MSB2LSB;
        td->td_bitspersample = 1;
        td->td_threshholding = THRESHHOLD_BILEVEL;
        td->td_orientation = ORIENTATION_TOPLEFT;
        td->td_samplesperpixel = 1;
        td->td_rowsperstrip = (uint32) -1;
        td->td_tilewidth = (uint32) -1;
        td->td_tilelength = (uint32) -1;
        td->td_tiledepth = 1;
        td->td_resolutionunit = RESUNIT_INCH;
        td->td_sampleformat = SAMPLEFORMAT_VOID;
        td->td_imagedepth = 1;
#ifdef YCBCR_SUPPORT
        td->td_ycbcrsubsampling[0] = 2;
        td->td_ycbcrsubsampling[1] = 2;
        td->td_ycbcrpositioning = YCBCRPOSITION_CENTERED;
#endif
#ifdef CMYK_SUPPORT
        td->td_inkset = INKSET_CMYK;
#endif
        tif->tif_postdecode = _TIFFNoPostDecode;
        tif->tif_vsetfield = _TIFFVSetField;
        tif->tif_vgetfield = _TIFFVGetField;
        tif->tif_printdir = NULL;
        /*
         *  Give client code a chance to install their own
         *  tag extensions & methods, prior to compression overloads.
         */
        if (_TIFFextender)
                (*_TIFFextender)(tif);
        (void) TIFFSetField(tif, TIFFTAG_COMPRESSION, COMPRESSION_NONE);
        /*
         * NB: The directory is marked dirty as a result of setting
         * up the default compression scheme.  However, this really
         * isn't correct -- we want TIFF_DIRTYDIRECT to be set only
```

```
         * if the user does something.  We could just do the setup
         * by hand, but it seems better to use the normal mechanism
         * (i.e. TIFFSetField).
         */
        tif->tif_flags &= ~TIFF_DIRTYDIRECT;
        return (1);
}

static int
```

---

kfax'TIFFAdvanceDirectory() (./kdegraphics/kfax/libtiffax/tif_dir.c:826)

```
TIFFAdvanceDirectory(TIFF* tif, uint32* nextdir, toff_t* off)
{
        static const char module[] = "TIFFAdvanceDirectory";
        uint16 dircount;

        if (!SeekOK(tif, *nextdir) ||
            !ReadOK(tif, &dircount, sizeof (uint16))) {
                TIFFError(module, "%s: Error fetching directory count",
                    tif->tif_name);
                return (0);
        }
        if (tif->tif_flags & TIFF_SWAB)
                TIFFSwabShort(&dircount);
        if (off != NULL)
                *off = TIFFSeekFile(tif,
                    dircount*sizeof (TIFFDirEntry), SEEK_CUR);
        else
                (void) TIFFSeekFile(tif,
                    dircount*sizeof (TIFFDirEntry), SEEK_CUR);
        if (!ReadOK(tif, nextdir, sizeof (uint32))) {
                TIFFError(module, "%s: Error fetching directory link",
                    tif->tif_name);
                return (0);
        }
        if (tif->tif_flags & TIFF_SWAB)
                TIFFSwabLong(nextdir);
        return (1);
}

/*
 * Set the n-th directory as the current directory.
 * NB: Directories are numbered starting at 0.
 */
int
```

---

kfax'TIFFSetDirectory() (./kdegraphics/kfax/libtiffax/tif_dir.c:860)

```
TIFFSetDirectory(TIFF* tif, tdir_t dirn)
{
        uint32 nextdir;
        tdir_t n;

        nextdir = tif->tif_header.tiff_diroff;
        for (n = dirn; n > 0 && nextdir != 0; n--)
```

```
                if (!TIFFAdvanceDirectory(tif, &nextdir, NULL))
                        return (0);
        tif->tif_nextdiroff = nextdir;
        /*
         * Set curdir to the actual directory index.  The
         * -1 is because TIFFReadDirectory will increment
         * tif_curdir after successfully reading the directory.
         */
        tif->tif_curdir = (dirn - n) - 1;
        return (TIFFReadDirectory(tif));
}

/*
 * Set the current directory to be the directory
 * located at the specified file offset.  This interface
 * is used mainly to access directories linked with
 * the SubIFD tag (e.g. thumbnail images).
 */
int
```

---

kfax'TIFFSetSubDirectory() (./kdegraphics/kfax/libtiffax/tif_dir.c:886)

```
TIFFSetSubDirectory(TIFF* tif, uint32 diroff)
{
        tif->tif_nextdiroff = diroff;
        return (TIFFReadDirectory(tif));
}

/*
 * Return file offset of the current directory.
 */
uint32
```

---

kfax'TIFFCurrentDirOffset() (./kdegraphics/kfax/libtiffax/tif_dir.c:896)

```
TIFFCurrentDirOffset(TIFF* tif)
{
        return (tif->tif_diroff);
}

/*
 * Return an indication of whether or not we are
 * at the last directory in the file.
 */
int
```

---

kfax'TIFFLastDirectory() (./kdegraphics/kfax/libtiffax/tif_dir.c:906)

```
TIFFLastDirectory(TIFF* tif)
{
        return (tif->tif_nextdiroff == 0);
}
```

```
/*
 * Unlink the specified directory from the directory chain.
 */
int
```

---

kfax'TIFFUnlinkDirectory() (./kdegraphics/kfax/libtiffax/tif_dir.c:915)

```
TIFFUnlinkDirectory(TIFF* tif, tdir_t dirn)
{
        static const char module[] = "TIFFUnlinkDirectory";
        uint32 nextdir;
        toff_t off;
        tdir_t n;

        if (tif->tif_mode == O_RDONLY) {
                TIFFError(module, "Can not unlink directory in read-only file");
                return (0);
        }
        /*
         * Go to the directory before the one we want
         * to unlink and nab the offset of the link
         * field we'll need to patch.
         */
        nextdir = tif->tif_header.tiff_diroff;
        off = sizeof (uint16) + sizeof (uint16);
        for (n = dirn-1; n > 0; n--) {
                if (nextdir == 0) {
                        TIFFError(module, "Directory %d does not exist", dirn);
                        return (0);
                }
                if (!TIFFAdvanceDirectory(tif, &nextdir, &off))
                        return (0);
        }
        /*
         * Advance to the directory to be unlinked and fetch
         * the offset of the directory that follows.
         */
        if (!TIFFAdvanceDirectory(tif, &nextdir, NULL))
                return (0);
        /*
         * Go back and patch the link field of the preceding
         * directory to point to the offset of the directory
         * that follows.
         */
        (void) TIFFSeekFile(tif, off, SEEK_SET);
        if (tif->tif_flags & TIFF_SWAB)
                TIFFSwabLong(&nextdir);
        if (!WriteOK(tif, &nextdir, sizeof (uint32))) {
                TIFFError(module, "Error writing directory link");
                return (0);
        }
        /*
         * Leave directory state setup safely.  We don't have
         * facilities for doing inserting and removing directories,
         * so it's safest to just invalidate everything.  This
         * means that the caller can only append to the directory
         * chain.
         */
```

```
        (*tif->tif_cleanup)(tif);
        if ((tif->tif_flags & TIFF_MYBUFFER) && tif->tif_rawdata) {
                _TIFFfree(tif->tif_rawdata);
                tif->tif_rawdata = NULL;
                tif->tif_rawcc = 0;
        }
        tif->tif_flags &= ~(TIFF_BEENWRITING|TIFF_BUFFERSETUP|TIFF_POSTENCODE);
        TIFFFreeDirectory(tif);
        TIFFDefaultDirectory(tif);
        tif->tif_diroff = 0;                    /* force link on next write */
        tif->tif_nextdiroff = 0;                /* next write must be at end */
        tif->tif_curoff = 0;
        tif->tif_row = (uint32) -1;
        tif->tif_curstrip = (tstrip_t) -1;
        return (1);
}
```

---

kfax'_TIFFSetupFieldInfo() (./kdegraphics/kfax/libtiffax/tif_dirinfo.c:221)

```
_TIFFSetupFieldInfo(TIFF* tif)
{
        if (tif->tif_fieldinfo) {
                _TIFFfree(tif->tif_fieldinfo);
                tif->tif_nfields = 0;
        }
        _TIFFMergeFieldInfo(tif, tiffFieldInfo, N(tiffFieldInfo));
}

static int
```

---

kfax'tagCompare() (./kdegraphics/kfax/libtiffax/tif_dirinfo.c:231)

```
tagCompare(const void* a, const void* b)
{
        const TIFFFieldInfo* ta = *(const TIFFFieldInfo**) a;
        const TIFFFieldInfo* tb = *(const TIFFFieldInfo**) b;
        int c = ta->field_tag - tb->field_tag;
        return (c != 0 ? c : tb->field_type - ta->field_type);
}

void
```

---

kfax'_TIFFMergeFieldInfo() (./kdegraphics/kfax/libtiffax/tif_dirinfo.c:240)

```
_TIFFMergeFieldInfo(TIFF* tif, const TIFFFieldInfo info[], int n)
{
        TIFFFieldInfo** tp;
        int i;

        if (tif->tif_nfields > 0) {
                tif->tif_fieldinfo = (TIFFFieldInfo**)
                    _TIFFrealloc(tif->tif_fieldinfo,
                        (tif->tif_nfields+n) * sizeof (TIFFFieldInfo*));
```

```
        } else {
                tif->tif_fieldinfo = (TIFFFieldInfo**)
                    _TIFFmalloc(n * sizeof (TIFFFieldInfo*));
        }
        tp = &tif->tif_fieldinfo[tif->tif_nfields];
        for (i = 0; i < n; i++)
                tp[i] = (TIFFFieldInfo*) &info[i];      /* XXX */
        /*
         * NB: the core tags are presumed sorted correctly.
         */
        if (tif->tif_nfields > 0)
                qsort(tif->tif_fieldinfo, (size_t) (tif->tif_nfields += n),
                    sizeof (TIFFFieldInfo*), tagCompare);
        else
                tif->tif_nfields += n;
}

void
```

---

kfax'_TIFFPrintFieldInfo() (./kdegraphics/kfax/libtiffax/tif_dirinfo.c:267)

```
_TIFFPrintFieldInfo(TIFF* tif, FILE* fd)
{
        int i;

        fprintf(fd, "%s: \n", tif->tif_name);
        for (i = 0; i < tif->tif_nfields; i++) {
                const TIFFFieldInfo* fip = tif->tif_fieldinfo[i];
                fprintf(fd, "field[%2d] %5u, %2d, %2d, %d, %2d, %5s, %5s, %s\n"
                        , i
                        , fip->field_tag
                        , fip->field_readcount, fip->field_writecount
                        , fip->field_type
                        , fip->field_bit
                        , fip->field_oktochange ? "TRUE" : "FALSE"
                        , fip->field_passcount ? "TRUE" : "FALSE"
                        , fip->field_name
                );
        }
}
```

---

kfax'_TIFFSampleToTagType() (./kdegraphics/kfax/libtiffax/tif_dirinfo.c:307)

```
_TIFFSampleToTagType(TIFF* tif)
{
        int bps = (int) TIFFhowmany(tif->tif_dir.td_bitspersample, 8);

        switch (tif->tif_dir.td_sampleformat) {
        case SAMPLEFORMAT_IEEEFP:
                return (bps == 4 ? TIFF_FLOAT : TIFF_DOUBLE);
        case SAMPLEFORMAT_INT:
                return (bps <= 1 ? TIFF_SBYTE :
                    bps <= 2 ? TIFF_SSHORT : TIFF_SLONG);
        case SAMPLEFORMAT_UINT:
                return (bps <= 1 ? TIFF_BYTE :
```

```
                bps <= 2 ? TIFF_SHORT : TIFF_LONG);
        case SAMPLEFORMAT_VOID:
                return (TIFF_UNDEFINED);
        }
        /*NOTREACHED*/
        return (TIFF_UNDEFINED);
}

const TIFFFieldInfo*
```

---

kfax'_TIFFFindFieldInfo() (./kdegraphics/kfax/libtiffax/tif_dirinfo.c:328)

```
_TIFFFindFieldInfo(TIFF* tif, ttag_t tag, TIFFDataType dt)
{
        static const TIFFFieldInfo *last = NULL;
        int i, n;

        if (last && last->field_tag == tag &&
            (dt == TIFF_ANY || dt == last->field_type))
                return (last);
        /* NB: if table gets big, use sorted search (e.g. binary search) */
        for (i = 0, n = tif->tif_nfields; i < n; i++) {
                const TIFFFieldInfo* fip = tif->tif_fieldinfo[i];
                if (fip->field_tag == tag &&
                    (dt == TIFF_ANY || fip->field_type == dt))
                        return (last = fip);
        }
        return ((const TIFFFieldInfo *)0);
}

#include <assert.h>
#include <stdio.h>

const TIFFFieldInfo*
```

---

kfax'_TIFFFieldWithTag() (./kdegraphics/kfax/libtiffax/tif_dirinfo.c:350)

```
_TIFFFieldWithTag(TIFF* tif, ttag_t tag)
{
        const TIFFFieldInfo* fip = _TIFFFindFieldInfo(tif, tag, TIFF_ANY);
        if (!fip) {
                TIFFError("TIFFFieldWithTag",
                    "Internal error, unknown tag 0x%x", (u_int) tag);
                assert(fip != NULL);
                /*NOTREACHED*/
        }
        return (fip);
}
```

---

kfax'CheckMalloc() (./kdegraphics/kfax/libtiffax/tif_dirread.c:67)

```
CheckMalloc(TIFF* tif, tsize_t n, const char* what)
{
```

```
        char *cp = (char*)_TIFFmalloc(n);
        if (cp == NULL)
                TIFFError(tif->tif_name, "No space %s", what);
        return (cp);
}

/*
 * Read the next TIFF directory from a file
 * and convert it to the internal format.
 * We read directories sequentially.
 */
int
```

---

kfax'TIFFReadDirectory() (./kdegraphics/kfax/libtiffax/tif_dirread.c:81)

```
TIFFReadDirectory(TIFF* tif)
{
        register TIFFDirEntry* dp;
        register int n;
        register TIFFDirectory* td;
        TIFFDirEntry* dir;
        int iv;
        long v;
        double dv;
        const TIFFFieldInfo* fip;
        int fix;
        uint16 dircount;
        uint32 nextdiroff;
        char* cp;
        int diroutoforderwarning = 0;

        tif->tif_diroff = tif->tif_nextdiroff;
        if (tif->tif_diroff == 0)               /* no more directories */
                return (0);
        /*
         * Cleanup any previous compression state.
         */
        if (tif->tif_curdir != (tdir_t) -1)
                (*tif->tif_cleanup)(tif);
        tif->tif_curdir++;
        nextdiroff = 0;
        if (!isMapped(tif)) {
                if (!SeekOK(tif, tif->tif_diroff)) {
                        TIFFError(tif->tif_name,
                            "Seek error accessing TIFF directory");
                        return (0);
                }
                if (!ReadOK(tif, &dircount, sizeof (uint16))) {
                        TIFFError(tif->tif_name,
                            "Can not read TIFF directory count");
                        return (0);
                }
                if (tif->tif_flags & TIFF_SWAB)
                        TIFFSwabShort(&dircount);
                dir = (TIFFDirEntry *)CheckMalloc(tif,
                    dircount * sizeof (TIFFDirEntry), "to read TIFF directory");
                if (dir == NULL)
                        return (0);
```

```
		if (!ReadOK(tif, dir, dircount*sizeof (TIFFDirEntry))) {
			TIFFError(tif->tif_name, "Can not read TIFF directory");
			goto bad;
		}
		/*
		 * Read offset to next directory for sequential scans.
		 */
		(void) ReadOK(tif, &nextdiroff, sizeof (uint32));
	} else {
		toff_t off = tif->tif_diroff;

		if (off + sizeof (short) > tif->tif_size) {
			TIFFError(tif->tif_name,
			    "Can not read TIFF directory count");
			return (0);
		} else
			_TIFFmemcpy(&dircount, tif->tif_base + off, sizeof (uint
		off += sizeof (uint16);
		if (tif->tif_flags & TIFF_SWAB)
			TIFFSwabShort(&dircount);
		dir = (TIFFDirEntry *)CheckMalloc(tif,
		    dircount * sizeof (TIFFDirEntry), "to read TIFF directory");
		if (dir == NULL)
			return (0);
		if (off + dircount*sizeof (TIFFDirEntry) > tif->tif_size) {
			TIFFError(tif->tif_name, "Can not read TIFF directory");
			goto bad;
		} else
			_TIFFmemcpy(dir, tif->tif_base + off,
			    dircount*sizeof (TIFFDirEntry));
		off += dircount* sizeof (TIFFDirEntry);
		if (off + sizeof (uint32) < tif->tif_size)
			_TIFFmemcpy(&nextdiroff, tif->tif_base+off, sizeof (uint
	}
	if (tif->tif_flags & TIFF_SWAB)
		TIFFSwabLong(&nextdiroff);
	tif->tif_nextdiroff = nextdiroff;

	tif->tif_flags &= ~TIFF_BEENWRITING;	/* reset before new dir */
	/*
	 * Setup default value and then make a pass over
	 * the fields to check type and tag information,
	 * and to extract info required to size data
	 * structures.  A second pass is made afterwards
	 * to read in everthing not taken in the first pass.
	 */
	td = &tif->tif_dir;
	/* free any old stuff and reinit */
	TIFFFreeDirectory(tif);
	TIFFDefaultDirectory(tif);
	/*
	 * Electronic Arts writes gray-scale TIFF files
	 * without a PlanarConfiguration directory entry.
	 * Thus we setup a default value here, even though
	 * the TIFF spec says there is no default value.
	 */
	TIFFSetField(tif, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);

	/*
	 * Sigh, we must make a separate pass through the
	 * directory for the following reason:
```

```
         *
         * We must process the Compression tag in the first pass
         * in order to merge in codec-private tag definitions (otherwise
         * we may get complaints about unknown tags).  However, the
         * Compression tag may be dependent on the SamplesPerPixel
         * tag value because older TIFF specs permited Compression
         * to be written as a SamplesPerPixel-count tag entry.
         * Thus if we don't first figure out the correct SamplesPerPixel
         * tag value then we may end up ignoring the Compression tag
         * value because it has an incorrect count value (if the
         * true value of SamplesPerPixel is not 1).
         *
         * It sure would have been nice if Aldus had really thought
         * this stuff through carefully.
         */
        for (dp = dir, n = dircount; n > 0; n--, dp++) {
                if (tif->tif_flags & TIFF_SWAB) {
                        TIFFSwabArrayOfShort(&dp->tdir_tag, 2);
                        TIFFSwabArrayOfLong(&dp->tdir_count, 2);
                }
                if (dp->tdir_tag == TIFFTAG_SAMPLESPERPIXEL) {
                        if (!TIFFFetchNormalTag(tif, dp))
                                goto bad;
                        dp->tdir_tag = IGNORE;
                }
        }
        /*
         * First real pass over the directory.
         */
        fix = 0;
        for (dp = dir, n = dircount; n > 0; n--, dp++) {
                /*
                 * Find the field information entry for this tag.
                 */
                if (dp->tdir_tag == IGNORE)
                        continue;
                /*
                 * Silicon Beach (at least) writes unordered
                 * directory tags (violating the spec).  Handle
                 * it here, but be obnoxious (maybe they'll fix it?).
                 */
                if (dp->tdir_tag < tif->tif_fieldinfo[fix]->field_tag) {
                        if (!diroutoforderwarning) {
                                TIFFWarning(tif->tif_name,
        "invalid TIFF directory; tags are not sorted in ascending order");
                                diroutoforderwarning = 1;
                        }
                        fix = 0;                        /* O(n^2) */
                }
                while (fix < tif->tif_nfields &&
                    tif->tif_fieldinfo[fix]->field_tag < dp->tdir_tag)
                        fix++;
                if (fix == tif->tif_nfields ||
                    tif->tif_fieldinfo[fix]->field_tag != dp->tdir_tag) {
                        TIFFWarning(tif->tif_name,
                            "unknown field with tag %d (0x%x) ignored",
                            dp->tdir_tag,  dp->tdir_tag);
                        dp->tdir_tag = IGNORE;
                        fix = 0;                        /* restart search */
                        continue;
                }
```

```
		/*
		 * Null out old tags that we ignore.
		 */
		if (tif->tif_fieldinfo[fix]->field_bit == FIELD_IGNORE) {
	ignore:
			dp->tdir_tag = IGNORE;
			continue;
		}
		/*
		 * Check data type.
		 */
		fip = tif->tif_fieldinfo[fix];
		while (dp->tdir_type != (u_short) fip->field_type) {
			if (fip->field_type == TIFF_ANY)	/* wildcard */
				break;
			fip++, fix++;
			if (fix == tif->tif_nfields ||
			    fip->field_tag != dp->tdir_tag) {
				TIFFWarning(tif->tif_name,
				   "wrong data type %d for \"%s\"; tag ignored",
				    dp->tdir_type, fip[-1].field_name);
				goto ignore;
			}
		}
		/*
		 * Check count if known in advance.
		 */
		if (fip->field_readcount != TIFF_VARIABLE) {
			uint32 expected = (fip->field_readcount == TIFF_SPP) ?
			    (uint32) td->td_samplesperpixel :
			    (uint32) fip->field_readcount;
			if (!CheckDirCount(tif, dp, expected))
				goto ignore;
		}

		switch (dp->tdir_tag) {
		case TIFFTAG_COMPRESSION:
			/*
			 * The 5.0 spec says the Compression tag has
			 * one value, while earlier specs say it has
			 * one value per sample.  Because of this, we
			 * accept the tag if one value is supplied.
			 */
			if (dp->tdir_count == 1) {
				v = TIFFExtractData(tif,
				    dp->tdir_type, dp->tdir_offset);
				if (!TIFFSetField(tif, dp->tdir_tag, (int)v))
					goto bad;
				break;
			}
			if (!TIFFFetchPerSampleShorts(tif, dp, &iv) ||
			    !TIFFSetField(tif, dp->tdir_tag, iv))
				goto bad;
			dp->tdir_tag = IGNORE;
			break;
		case TIFFTAG_STRIPOFFSETS:
		case TIFFTAG_STRIPBYTECOUNTS:
		case TIFFTAG_TILEOFFSETS:
		case TIFFTAG_TILEBYTECOUNTS:
			TIFFSetFieldBit(tif, fip->field_bit);
			break;
```

```
        case TIFFTAG_IMAGEWIDTH:
        case TIFFTAG_IMAGELENGTH:
        case TIFFTAG_IMAGEDEPTH:
        case TIFFTAG_TILELENGTH:
        case TIFFTAG_TILEWIDTH:
        case TIFFTAG_TILEDEPTH:
        case TIFFTAG_PLANARCONFIG:
        case TIFFTAG_ROWSPERSTRIP:
                if (!TIFFFetchNormalTag(tif, dp))
                        goto bad;
                dp->tdir_tag = IGNORE;
                break;
        case TIFFTAG_EXTRASAMPLES:
                (void) TIFFFetchExtraSamples(tif, dp);
                dp->tdir_tag = IGNORE;
                break;
        }
}

/*
 * Allocate directory structure and setup defaults.
 */
if (!TIFFFieldSet(tif, FIELD_IMAGEDIMENSIONS)) {
        MissingRequired(tif, "ImageLength");
        goto bad;
}
if (!TIFFFieldSet(tif, FIELD_PLANARCONFIG)) {
        MissingRequired(tif, "PlanarConfiguration");
        goto bad;
}
/*
 * Setup appropriate structures (by strip or by tile)
 */
if (!TIFFFieldSet(tif, FIELD_TILEDIMENSIONS)) {
        td->td_nstrips = TIFFNumberOfStrips(tif);
        td->td_tilewidth = td->td_imagewidth;
        td->td_tilelength = td->td_rowsperstrip;
        td->td_tiledepth = td->td_imagedepth;
        tif->tif_flags &= ~TIFF_ISTILED;
} else {
        td->td_nstrips = TIFFNumberOfTiles(tif);
        tif->tif_flags |= TIFF_ISTILED;
}
td->td_stripsperimage = td->td_nstrips;
if (td->td_planarconfig == PLANARCONFIG_SEPARATE)
        td->td_stripsperimage /= td->td_samplesperpixel;
if (!TIFFFieldSet(tif, FIELD_STRIPOFFSETS)) {
        MissingRequired(tif,
            isTiled(tif) ? "TileOffsets" : "StripOffsets");
        goto bad;
}

/*
 * Second pass: extract other information.
 */
for (dp = dir, n = dircount; n > 0; n--, dp++) {
        if (dp->tdir_tag == IGNORE)
                continue;
        switch (dp->tdir_tag) {
        case TIFFTAG_MINSAMPLEVALUE:
        case TIFFTAG_MAXSAMPLEVALUE:
```

```
        case TIFFTAG_BITSPERSAMPLE:
                /*
                 * The 5.0 spec says the Compression tag has
                 * one value, while earlier specs say it has
                 * one value per sample.  Because of this, we
                 * accept the tag if one value is supplied.
                 *
                 * The MinSampleValue, MaxSampleValue and
                 * BitsPerSample tags are supposed to be written
                 * as one value/sample, but some vendors incorrectly
                 * write one value only -- so we accept that
                 * as well (yech).
                 */
                if (dp->tdir_count == 1) {
                        v = TIFFExtractData(tif,
                            dp->tdir_type, dp->tdir_offset);
                        if (!TIFFSetField(tif, dp->tdir_tag, (int)v))
                                goto bad;
                        break;
                }
                /* fall thru... */
        case TIFFTAG_DATATYPE:
        case TIFFTAG_SAMPLEFORMAT:
                if (!TIFFFetchPerSampleShorts(tif, dp, &iv) ||
                    !TIFFSetField(tif, dp->tdir_tag, iv))
                        goto bad;
                break;
        case TIFFTAG_SMINSAMPLEVALUE:
        case TIFFTAG_SMAXSAMPLEVALUE:
                if (!TIFFFetchPerSampleAnys(tif, dp, &dv) ||
                    !TIFFSetField(tif, dp->tdir_tag, dv))
                        goto bad;
                break;
        case TIFFTAG_STRIPOFFSETS:
        case TIFFTAG_TILEOFFSETS:
                if (!TIFFFetchStripThing(tif, dp,
                    td->td_nstrips, &td->td_stripoffset))
                        goto bad;
                break;
        case TIFFTAG_STRIPBYTECOUNTS:
        case TIFFTAG_TILEBYTECOUNTS:
                if (!TIFFFetchStripThing(tif, dp,
                    td->td_nstrips, &td->td_stripbytecount))
                        goto bad;
                break;
        case TIFFTAG_COLORMAP:
        case TIFFTAG_TRANSFERFUNCTION:
                /*
                 * TransferFunction can have either 1x or 3x data
                 * values; Colormap can have only 3x items.
                 */
                v = 1L<<td->td_bitspersample;
                if (dp->tdir_tag == TIFFTAG_COLORMAP ||
                    dp->tdir_count != (uint32) v) {
                        if (!CheckDirCount(tif, dp, (uint32)(3*v)))
                                break;
                }
                v *= sizeof (uint16);
                cp = CheckMalloc(tif, dp->tdir_count * sizeof (uint16),
                    "to read \"TransferFunction\" tag");
                if (cp != NULL) {
```

```
                                if (TIFFFetchData(tif, dp, cp)) {
                                        /*
                                         * This deals with there being only
                                         * one array to apply to all samples.
                                         */
                                        uint32 c =
                                            (uint32)1 << td->td_bitspersample;
                                        if (dp->tdir_count == c)
                                                v = 0;
                                        TIFFSetField(tif, dp->tdir_tag,
                                            cp, cp+v, cp+2*v);
                                }
                                _TIFFfree(cp);
                        }
                        break;
                case TIFFTAG_PAGENUMBER:
                case TIFFTAG_HALFTONEHINTS:
                case TIFFTAG_YCBCRSUBSAMPLING:
                case TIFFTAG_DOTRANGE:
                        (void) TIFFFetchShortPair(tif, dp);
                        break;
#ifdef COLORIMETRY_SUPPORT
                case TIFFTAG_REFERENCEBLACKWHITE:
                        (void) TIFFFetchRefBlackWhite(tif, dp);
                        break;
#endif
/* BEGIN REV 4.0 COMPATIBILITY */
                case TIFFTAG_OSUBFILETYPE:
                        v = 0;
                        switch (TIFFExtractData(tif, dp->tdir_type,
                            dp->tdir_offset)) {
                        case OFILETYPE_REDUCEDIMAGE:
                                v = FILETYPE_REDUCEDIMAGE;
                                break;
                        case OFILETYPE_PAGE:
                                v = FILETYPE_PAGE;
                                break;
                        }
                        if (v)
                                (void) TIFFSetField(tif,
                                    TIFFTAG_SUBFILETYPE, (int)v);
                        break;
/* END REV 4.0 COMPATIBILITY */
                default:
                        (void) TIFFFetchNormalTag(tif, dp);
                        break;
                }
        }
        /*
         * Verify Palette image has a Colormap.
         */
        if (td->td_photometric == PHOTOMETRIC_PALETTE &&
            !TIFFFieldSet(tif, FIELD_COLORMAP)) {
                MissingRequired(tif, "Colormap");
                goto bad;
        }
        /*
         * Attempt to deal with a missing StripByteCounts tag.
         */
        if (!TIFFFieldSet(tif, FIELD_STRIPBYTECOUNTS)) {
                /*
```

```
                 * Some manufacturers violate the spec by not giving
                 * the size of the strips.  In this case, assume there
                 * is one uncompressed strip of data.
                 */
                if ((td->td_planarconfig == PLANARCONFIG_CONTIG &&
                    td->td_nstrips > 1) ||
                    (td->td_planarconfig == PLANARCONFIG_SEPARATE &&
                     td->td_nstrips != td->td_samplesperpixel)) {
                    MissingRequired(tif, "StripByteCounts");
                    goto bad;
                }
                TIFFWarning(tif->tif_name,
"TIFF directory is missing required \"%s\" field, calculating from imagelength",
                    _TIFFFieldWithTag(tif,TIFFTAG_STRIPBYTECOUNTS)->field_name);
                EstimateStripByteCounts(tif, dir, dircount);
```

---

kfax'EstimateStripByteCounts() (./kdegraphics/kfax/libtiffax/tif_dirread.c:562)

```
EstimateStripByteCounts(TIFF* tif, TIFFDirEntry* dir, uint16 dircount)
{
        register TIFFDirEntry *dp;
        register TIFFDirectory *td = &tif->tif_dir;
        uint16 i;

        if (td->td_stripbytecount)
                _TIFFfree(td->td_stripbytecount);
        td->td_stripbytecount = (uint32*)
            CheckMalloc(tif, td->td_nstrips * sizeof (uint32),
                "for \"StripByteCounts\" array");
        if (td->td_compression != COMPRESSION_NONE) {
                uint32 space = (uint32)(sizeof (TIFFHeader)
                    + sizeof (uint16)
                    + (dircount * sizeof (TIFFDirEntry))
                    + sizeof (uint32));
                toff_t filesize = TIFFGetFileSize(tif);
                uint16 n;

                /* calculate amount of space used by indirect values */
                for (dp = dir, n = dircount; n > 0; n--, dp++) {
                        uint32 cc = dp->tdir_count*tiffDataWidth[dp->tdir_type];
                        if (cc > sizeof (uint32))
                                space += cc;
                }
                space = (filesize - space) / td->td_samplesperpixel;
                for (i = 0; i < td->td_nstrips; i++)
                        td->td_stripbytecount[i] = space;
                /*
                 * This gross hack handles the case were the offset to
                 * the last strip is past the place where we think the strip
                 * should begin.  Since a strip of data must be contiguous,
                 * it's safe to assume that we've overestimated the amount
                 * of data in the strip and trim this number back accordingly.
                 */
                i--;
                if (td->td_stripoffset[i] + td->td_stripbytecount[i] > filesize)
                        td->td_stripbytecount[i] =
                            filesize - td->td_stripoffset[i];
        } else {
```

```
                uint32 rowbytes = TIFFScanlineSize(tif);
                uint32 rowsperstrip = td->td_imagelength / td->td_nstrips;
                for (i = 0; i < td->td_nstrips; i++)
                        td->td_stripbytecount[i] = rowbytes*rowsperstrip;
        }
        TIFFSetFieldBit(tif, FIELD_STRIPBYTECOUNTS);
        if (!TIFFFieldSet(tif, FIELD_ROWSPERSTRIP))
                td->td_rowsperstrip = td->td_imagelength;
}

static void
```

---

kfax'MissingRequired() (./kdegraphics/kfax/libtiffax/tif_dirread.c:613)

```
MissingRequired(TIFF* tif, const char* tagname)
{
        TIFFError(tif->tif_name,
            "TIFF directory is missing required \"%s\" field", tagname);
}

/*
 * Check the count field of a directory
 * entry against a known value.  The caller
 * is expected to skip/ignore the tag if
 * there is a mismatch.
 */
static int
```

---

kfax'CheckDirCount() (./kdegraphics/kfax/libtiffax/tif_dirread.c:626)

```
CheckDirCount(TIFF* tif, TIFFDirEntry* dir, uint32 count)
{
        if (count != dir->tdir_count) {
                TIFFWarning(tif->tif_name,
        "incorrect count for field \"%s\" (%lu, expecting %lu); tag ignored",
                    _TIFFFieldWithTag(tif, dir->tdir_tag)->field_name,
                    dir->tdir_count, count);
                return (0);
        }
        return (1);
}

/*
 * Fetch a contiguous directory item.
 */
static tsize_t
```

---

kfax'TIFFFetchData() (./kdegraphics/kfax/libtiffax/tif_dirread.c:642)

```
TIFFFetchData(TIFF* tif, TIFFDirEntry* dir, char* cp)
{
        int w = tiffDataWidth[dir->tdir_type];
        tsize_t cc = dir->tdir_count * w;
```

```
        if (!isMapped(tif)) {
                if (!SeekOK(tif, dir->tdir_offset))
                        goto bad;
                if (!ReadOK(tif, cp, cc))
                        goto bad;
        } else {
                if (dir->tdir_offset + cc > tif->tif_size)
                        goto bad;
                _TIFFmemcpy(cp, tif->tif_base + dir->tdir_offset, cc);
        }
        if (tif->tif_flags & TIFF_SWAB) {
                switch (dir->tdir_type) {
                case TIFF_SHORT:
                case TIFF_SSHORT:
                        TIFFSwabArrayOfShort((uint16*) cp, dir->tdir_count);
                        break;
                case TIFF_LONG:
                case TIFF_SLONG:
                case TIFF_FLOAT:
                        TIFFSwabArrayOfLong((uint32*) cp, dir->tdir_count);
                        break;
                case TIFF_RATIONAL:
                case TIFF_SRATIONAL:
                        TIFFSwabArrayOfLong((uint32*) cp, 2*dir->tdir_count);
                        break;
                case TIFF_DOUBLE:
                        TIFFSwabArrayOfDouble((double*) cp, dir->tdir_count);
                        break;
                }
        }
        return (cc);
bad:
        TIFFError(tif->tif_name, "Error fetching data for field \"%s\"",
            _TIFFFieldWithTag(tif, dir->tdir_tag)->field_name);
        return ((tsize_t) 0);
}

/*
 * Fetch an ASCII item from the file.
 */
static tsize_t
```

---

kfax'TIFFFetchString() (./kdegraphics/kfax/libtiffax/tif_dirread.c:688)

```
TIFFFetchString(TIFF* tif, TIFFDirEntry* dir, char* cp)
{
        if (dir->tdir_count <= 4) {
                uint32 l = dir->tdir_offset;
                if (tif->tif_flags & TIFF_SWAB)
                        TIFFSwabLong(&l);
                _TIFFmemcpy(cp, &l, dir->tdir_count);
                return (1);
        }
        return (TIFFFetchData(tif, dir, cp));
}

/*
```

```
 * Convert numerator+denominator to float.
 */
static int
```

---

kfax'cvtRational() (./kdegraphics/kfax/libtiffax/tif_dirread.c:704)

```
cvtRational(TIFF* tif, TIFFDirEntry* dir, uint32 num, uint32 denom, float* rv)
{
        if (denom == 0) {
                TIFFError(tif->tif_name,
                    "%s: Rational with zero denominator (num = %lu)",
                    _TIFFFieldWithTag(tif, dir->tdir_tag)->field_name, num);
                return (0);
        } else {
                if (dir->tdir_type == TIFF_RATIONAL)
                        *rv = ((float)num / (float)denom);
                else
                        *rv = ((float)(int32)num / (float)(int32)denom);
                return (1);
        }
}

/*
 * Fetch a rational item from the file
 * at offset off and return the value
 * as a floating point number.
 */
static float
```

---

kfax'TIFFFetchRational() (./kdegraphics/kfax/libtiffax/tif_dirread.c:726)

```
TIFFFetchRational(TIFF* tif, TIFFDirEntry* dir)
{
        uint32 l[2];
        float v;

        return (!TIFFFetchData(tif, dir, (char *)l) ||
            !cvtRational(tif, dir, l[0], l[1], &v) ? 1.0f : v);
}

/*
 * Fetch a single floating point value
 * from the offset field and return it
 * as a native float.
 */
static float
```

---

kfax'TIFFFetchFloat() (./kdegraphics/kfax/libtiffax/tif_dirread.c:741)

```
TIFFFetchFloat(TIFF* tif, TIFFDirEntry* dir)
{
        float v = (float)
            TIFFExtractData(tif, dir->tdir_type, dir->tdir_offset);
```

```
        TIFFCvtIEEEFloatToNative(tif, 1, &v);
        return (v);
}

/*
 * Fetch an array of BYTE or SBYTE values.
 */
static int
```

---

kfax'TIFFFetchByteArray() (./kdegraphics/kfax/libtiffax/tif_dirread.c:753)

```
TIFFFetchByteArray(TIFF* tif, TIFFDirEntry* dir, uint16* v)
{
        if (dir->tdir_count <= 4) {
                /*
                 * Extract data from offset field.
                 */
                if (tif->tif_header.tiff_magic == TIFF_BIGENDIAN) {
                        switch (dir->tdir_count) {
                        case 4: v[3] = dir->tdir_offset & 0xff;
                        case 3: v[2] = (dir->tdir_offset >> 8) & 0xff;
                        case 2: v[1] = (dir->tdir_offset >> 16) & 0xff;
                        case 1: v[0] = dir->tdir_offset >> 24;
                        }
                } else {
                        switch (dir->tdir_count) {
                        case 4: v[3] = dir->tdir_offset >> 24;
                        case 3: v[2] = (dir->tdir_offset >> 16) & 0xff;
                        case 2: v[1] = (dir->tdir_offset >> 8) & 0xff;
                        case 1: v[0] = dir->tdir_offset & 0xff;
                        }
                }
                return (1);
        } else
                return (TIFFFetchData(tif, dir, (char*) v) != 0);       /* XXX *
}

/*
 * Fetch an array of SHORT or SSHORT values.
 */
static int
```

---

kfax'TIFFFetchShortArray() (./kdegraphics/kfax/libtiffax/tif_dirread.c:783)

```
TIFFFetchShortArray(TIFF* tif, TIFFDirEntry* dir, uint16* v)
{
        if (dir->tdir_count <= 2) {
                if (tif->tif_header.tiff_magic == TIFF_BIGENDIAN) {
                        switch (dir->tdir_count) {
                        case 2: v[1] = dir->tdir_offset & 0xffff;
                        case 1: v[0] = dir->tdir_offset >> 16;
                        }
                } else {
                        switch (dir->tdir_count) {
                        case 2: v[1] = dir->tdir_offset >> 16;
                        case 1: v[0] = dir->tdir_offset & 0xffff;
```

```
                        }
                }
                return (1);
        } else
                return (TIFFFetchData(tif, dir, (char *)v) != 0);
}

/*
 * Fetch a pair of SHORT or BYTE values.
 */
static int
```

---

kfax'TIFFFetchShortPair() (./kdegraphics/kfax/libtiffax/tif_dirread.c:806)

```
TIFFFetchShortPair(TIFF* tif, TIFFDirEntry* dir)
{
        uint16 v[2];
        int ok = 0;

        switch (dir->tdir_type) {
        case TIFF_SHORT:
        case TIFF_SSHORT:
                ok = TIFFFetchShortArray(tif, dir, v);
                break;
        case TIFF_BYTE:
        case TIFF_SBYTE:
                ok  = TIFFFetchByteArray(tif, dir, v);
                break;
        }
        if (ok)
                TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
        return (ok);
}

/*
 * Fetch an array of LONG or SLONG values.
 */
static int
```

---

kfax'TIFFFetchLongArray() (./kdegraphics/kfax/libtiffax/tif_dirread.c:830)

```
TIFFFetchLongArray(TIFF* tif, TIFFDirEntry* dir, uint32* v)
{
        if (dir->tdir_count == 1) {
                v[0] = dir->tdir_offset;
                return (1);
        } else
                return (TIFFFetchData(tif, dir, (char*) v) != 0);
}

/*
 * Fetch an array of RATIONAL or SRATIONAL values.
 */
static int
```

---

kfax'TIFFFetchRationalArray() (./kdegraphics/kfax/libtiffax/tif_dirread.c:843)

```
TIFFFetchRationalArray(TIFF* tif, TIFFDirEntry* dir, float* v)
{
        int ok = 0;
        uint32* l;

        l = (uint32*)CheckMalloc(tif,
            dir->tdir_count*tiffDataWidth[dir->tdir_type],
            "to fetch array of rationals");
        if (l) {
                if (TIFFFetchData(tif, dir, (char *)l)) {
                        uint32 i;
                        for (i = 0; i < dir->tdir_count; i++) {
                                ok = cvtRational(tif, dir,
                                    l[2*i+0], l[2*i+1], &v[i]);
                                if (!ok)
                                        break;
                        }
                }
                _TIFFfree((char *)l);
        }
        return (ok);
}

/*
 * Fetch an array of FLOAT values.
 */
static int
```

---

kfax'TIFFFetchFloatArray() (./kdegraphics/kfax/libtiffax/tif_dirread.c:870)

```
TIFFFetchFloatArray(TIFF* tif, TIFFDirEntry* dir, float* v)
{

        if (dir->tdir_count == 1) {
                v[0] = *(float*) &dir->tdir_offset;
                TIFFCvtIEEEFloatToNative(tif, dir->tdir_count, v);
                return (1);
        } else  if (TIFFFetchData(tif, dir, (char*) v)) {
                TIFFCvtIEEEFloatToNative(tif, dir->tdir_count, v);
                return (1);
        } else
                return (0);
}

/*
 * Fetch an array of DOUBLE values.
 */
static int
```

---

kfax'TIFFFetchDoubleArray() (./kdegraphics/kfax/libtiffax/tif_dirread.c:888)

```
TIFFFetchDoubleArray(TIFF* tif, TIFFDirEntry* dir, double* v)
```

```
{
        if (TIFFFetchData(tif, dir, (char*) v)) {
                TIFFCvtIEEEDoubleToNative(tif, dir->tdir_count, v);
                return (1);
        } else
                return (0);
}

/*
 * Fetch an array of ANY values.  The actual values are
 * returned as doubles which should be able hold all the
 * types.  Yes, there really should be an tany_t to avoid
 * this potential non-portability ...  Note in particular
 * that we assume that the double return value vector is
 * large enough to read in any fundamental type.  We use
 * that vector as a buffer to read in the base type vector
 * and then convert it in place to double (from end
 * to front of course).
 */
static int
```

kfax'TIFFFetchAnyArray() (./kdegraphics/kfax/libtiffax/tif_dirread.c:909)

```
TIFFFetchAnyArray(TIFF* tif, TIFFDirEntry* dir, double* v)
{
        int i;

        switch (dir->tdir_type) {
        case TIFF_BYTE:
        case TIFF_SBYTE:
                if (!TIFFFetchByteArray(tif, dir, (uint16*) v))
                        return (0);
                if (dir->tdir_type == TIFF_BYTE) {
                        uint16* vp = (uint16*) v;
                        for (i = dir->tdir_count-1; i >= 0; i--)
                                v[i] = vp[i];
                } else {
                        int16* vp = (int16*) v;
                        for (i = dir->tdir_count-1; i >= 0; i--)
                                v[i] = vp[i];
                }
                break;
        case TIFF_SHORT:
        case TIFF_SSHORT:
                if (!TIFFFetchShortArray(tif, dir, (uint16*) v))
                        return (0);
                if (dir->tdir_type == TIFF_SHORT) {
                        uint16* vp = (uint16*) v;
                        for (i = dir->tdir_count-1; i >= 0; i--)
                                v[i] = vp[i];
                } else {
                        int16* vp = (int16*) v;
                        for (i = dir->tdir_count-1; i >= 0; i--)
                                v[i] = vp[i];
                }
                break;
        case TIFF_LONG:
        case TIFF_SLONG:
```

```
                if (!TIFFFetchLongArray(tif, dir, (uint32*) v))
                        return (0);
                if (dir->tdir_type == TIFF_LONG) {
                        uint32* vp = (uint32*) v;
                        for (i = dir->tdir_count-1; i >= 0; i--)
                                v[i] = vp[i];
                } else {
                        int32* vp = (int32*) v;
                        for (i = dir->tdir_count-1; i >= 0; i--)
                                v[i] = vp[i];
                }
                break;
        case TIFF_RATIONAL:
        case TIFF_SRATIONAL:
                if (!TIFFFetchRationalArray(tif, dir, (float*) v))
                        return (0);
                { float* vp = (float*) v;
                  for (i = dir->tdir_count-1; i >= 0; i--)
                        v[i] = vp[i];
                }
                break;
        case TIFF_FLOAT:
                if (!TIFFFetchFloatArray(tif, dir, (float*) v))
                        return (0);
                { float* vp = (float*) v;
                  for (i = dir->tdir_count-1; i >= 0; i--)
                        v[i] = vp[i];
                }
                break;
        case TIFF_DOUBLE:
                return (TIFFFetchDoubleArray(tif, dir, (double*) v));
        default:
                /* TIFF_NOTYPE */
                /* TIFF_ASCII */
                /* TIFF_UNDEFINED */
                TIFFError(tif->tif_name,
                    "Cannot read TIFF_ANY type %d for field \"%s\"",
                    _TIFFFieldWithTag(tif, dir->tdir_tag)->field_name);
                return (0);
        }
        return (1);
}

/*
 * Fetch a tag that is not handled by special case code.
 */
static int
```

---

kfax'TIFFFetchNormalTag() (./kdegraphics/kfax/libtiffax/tif_dirread.c:991)

```
TIFFFetchNormalTag(TIFF* tif, TIFFDirEntry* dp)
{
        static char mesg[] = "to fetch tag value";
        int ok = 0;
        const TIFFFieldInfo* fip = _TIFFFieldWithTag(tif, dp->tdir_tag);

        if (dp->tdir_count > 1) {               /* array of values */
                char* cp = NULL;
```

```
                switch (dp->tdir_type) {
                case TIFF_BYTE:
                case TIFF_SBYTE:
                        /* NB: always expand BYTE values to shorts */
                        cp = CheckMalloc(tif,
                            dp->tdir_count * sizeof (uint16), mesg);
                        ok = cp && TIFFFetchByteArray(tif, dp, (uint16*) cp);
                        break;
                case TIFF_SHORT:
                case TIFF_SSHORT:
                        cp = CheckMalloc(tif,
                            dp->tdir_count * sizeof (uint16), mesg);
                        ok = cp && TIFFFetchShortArray(tif, dp, (uint16*) cp);
                        break;
                case TIFF_LONG:
                case TIFF_SLONG:
                        cp = CheckMalloc(tif,
                            dp->tdir_count * sizeof (uint32), mesg);
                        ok = cp && TIFFFetchLongArray(tif, dp, (uint32*) cp);
                        break;
                case TIFF_RATIONAL:
                case TIFF_SRATIONAL:
                        cp = CheckMalloc(tif,
                            dp->tdir_count * sizeof (float), mesg);
                        ok = cp && TIFFFetchRationalArray(tif, dp, (float*) cp);
                        break;
                case TIFF_FLOAT:
                        cp = CheckMalloc(tif,
                            dp->tdir_count * sizeof (float), mesg);
                        ok = cp && TIFFFetchFloatArray(tif, dp, (float*) cp);
                        break;
                case TIFF_DOUBLE:
                        cp = CheckMalloc(tif,
                            dp->tdir_count * sizeof (double), mesg);
                        ok = cp && TIFFFetchDoubleArray(tif, dp, (double*) cp);
                        break;
                case TIFF_ASCII:
                case TIFF_UNDEFINED:            /* bit of a cheat... */
                        /*
                         * Some vendors write strings w/o the trailing
                         * NULL byte, so always append one just in case.
                         */
                        cp = CheckMalloc(tif, dp->tdir_count+1, mesg);
                        if (ok = (cp && TIFFFetchString(tif, dp, cp)))
                                cp[dp->tdir_count] = '\0';      /* XXX */
                        break;
                }
                if (ok) {
                        ok = (fip->field_passcount ?
                            TIFFSetField(tif, dp->tdir_tag, dp->tdir_count, cp)
                          : TIFFSetField(tif, dp->tdir_tag, cp));
                }
                if (cp != NULL)
                        _TIFFfree(cp);
        } else if (CheckDirCount(tif, dp, 1)) { /* singleton value */
                switch (dp->tdir_type) {
                case TIFF_BYTE:
                case TIFF_SBYTE:
                case TIFF_SHORT:
                case TIFF_SSHORT:
```

```
        /*
         * If the tag is also acceptable as a LONG or SLONG
         * then TIFFSetField will expect an uint32 parameter
         * passed to it (through varargs).  Thus, for machines
         * where sizeof (int) != sizeof (uint32) we must do
         * a careful check here.  It's hard to say if this
         * is worth optimizing.
         *
         * NB: We use TIFFFieldWithTag here knowing that
         *     it returns us the first entry in the table
         *     for the tag and that that entry is for the
         *     widest potential data type the tag may have.
         */
        { TIFFDataType type = fip->field_type;
          if (type != TIFF_LONG && type != TIFF_SLONG) {
                uint16 v = (uint16)
           TIFFExtractData(tif, dp->tdir_type, dp->tdir_offset);
                ok = (fip->field_passcount ?
                    TIFFSetField(tif, dp->tdir_tag, 1, &v)
                  : TIFFSetField(tif, dp->tdir_tag, v));
                break;
          }
        }
        /* fall thru... */
case TIFF_LONG:
case TIFF_SLONG:
        { uint32 v32 =
    TIFFExtractData(tif, dp->tdir_type, dp->tdir_offset);
          ok = (fip->field_passcount ?
              TIFFSetField(tif, dp->tdir_tag, 1, &v32)
            : TIFFSetField(tif, dp->tdir_tag, v32));
        }
        break;
case TIFF_RATIONAL:
case TIFF_SRATIONAL:
case TIFF_FLOAT:
        { float v = (dp->tdir_type == TIFF_FLOAT ?
              TIFFFetchFloat(tif, dp)
            : TIFFFetchRational(tif, dp));
          ok = (fip->field_passcount ?
              TIFFSetField(tif, dp->tdir_tag, 1, &v)
            : TIFFSetField(tif, dp->tdir_tag, v));
        }
        break;
case TIFF_DOUBLE:
        { double v;
          ok = (TIFFFetchDoubleArray(tif, dp, &v) &&
            (fip->field_passcount ?
              TIFFSetField(tif, dp->tdir_tag, 1, &v)
            : TIFFSetField(tif, dp->tdir_tag, v))
          );
        }
        break;
case TIFF_ASCII:
case TIFF_UNDEFINED:            /* bit of a cheat... */
        { char c[2];
          if (ok = (TIFFFetchString(tif, dp, c) != 0)) {
                c[1] = '\0';            /* XXX paranoid */
                ok = TIFFSetField(tif, dp->tdir_tag, c);
          }
        }
```

```
                        break;
                }
        }
        return (ok);
}
```

---

kfax'TIFFFetchPerSampleShorts() (./kdegraphics/kfax/libtiffax/tif_dirread.c:1134)

```
TIFFFetchPerSampleShorts(TIFF* tif, TIFFDirEntry* dir, int* pl)
{
        int samples = tif->tif_dir.td_samplesperpixel;
        int status = 0;

        if (CheckDirCount(tif, dir, (uint32) samples)) {
                uint16 buf[10];
                uint16* v = buf;

                if (samples > NITEMS(buf))
                        v = (uint16*) _TIFFmalloc(samples * sizeof (uint16));
                if (TIFFFetchShortArray(tif, dir, v)) {
                        int i;
                        for (i = 1; i < samples; i++)
                                if (v[i] != v[0]) {
                                        TIFFError(tif->tif_name,
                "Cannot handle different per-sample values for field \"%s\"",
                           _TIFFFieldWithTag(tif, dir->tdir_tag)->field_name);
                                        goto bad;
                                }
                        *pl = v[0];
                        status = 1;
                }
        bad:
                if (v != buf)
                        _TIFFfree((char*) v);
        }
        return (status);
}

/*
 * Fetch samples/pixel ANY values for
 * the specified tag and verify that
 * all values are the same.
 */
static int
```

---

kfax'TIFFFetchPerSampleAnys() (./kdegraphics/kfax/libtiffax/tif_dirread.c:1170)

```
TIFFFetchPerSampleAnys(TIFF* tif, TIFFDirEntry* dir, double* pl)
{
        int samples = (int) tif->tif_dir.td_samplesperpixel;
        int status = 0;
```

```
        if (CheckDirCount(tif, dir, (uint32) samples)) {
                double buf[10];
                double* v = buf;

                if (samples > NITEMS(buf))
                        v = (double*) _TIFFmalloc(samples * sizeof (double));
                if (TIFFFetchAnyArray(tif, dir, v)) {
                        int i;
                        for (i = 1; i < samples; i++)
                                if (v[i] != v[0]) {
                                        TIFFError(tif->tif_name,
                "Cannot handle different per-sample values for field \"%s\"",
                           _TIFFFieldWithTag(tif, dir->tdir_tag)->field_name);
                                        goto bad;
                                }
                        *pl = v[0];
                        status = 1;
                }
        bad:
                if (v != buf)
                        _TIFFfree(v);
        }
        return (status);
}
```

---

kfax'TIFFFetchStripThing() (./kdegraphics/kfax/libtiffax/tif_dirread.c:1207)

```
TIFFFetchStripThing(TIFF* tif, TIFFDirEntry* dir, long nstrips, uint32** lpp)
{
        register uint32* lp;
        int status;

        if (!CheckDirCount(tif, dir, (uint32) nstrips))
                return (0);
        /*
         * Allocate space for strip information.
         */
        if (*lpp == NULL &&
            (*lpp = (uint32 *)CheckMalloc(tif,
              nstrips * sizeof (uint32), "for strip array")) == NULL)
                return (0);
        lp = *lpp;
        if (dir->tdir_type == (int)TIFF_SHORT) {
                /*
                 * Handle uint16->uint32 expansion.
                 */
                uint16* dp = (uint16*) CheckMalloc(tif,
                    dir->tdir_count* sizeof (uint16), "to fetch strip tag");
                if (dp == NULL)
                        return (0);
                if (status = TIFFFetchShortArray(tif, dir, dp)) {
                        register uint16* wp = dp;
                        while (nstrips-- > 0)
                                *lp++ = *wp++;
                }
                _TIFFfree((char*) dp);
        } else
                status = TIFFFetchLongArray(tif, dir, lp);
```

```
        return (status);
}
```

---

kfax'TIFFFetchExtraSamples() (./kdegraphics/kfax/libtiffax/tif_dirread.c:1246)

```
TIFFFetchExtraSamples(TIFF* tif, TIFFDirEntry* dir)
{
        uint16 buf[10];
        uint16* v = buf;
        int status;

        if (dir->tdir_count > NITEMS(buf))
                v = (uint16*) _TIFFmalloc(dir->tdir_count * sizeof (uint16));
        if (dir->tdir_type == TIFF_BYTE)
                status = TIFFFetchByteArray(tif, dir, v);
        else
                status = TIFFFetchShortArray(tif, dir, v);
        if (status)
                status = TIFFSetField(tif, dir->tdir_tag, dir->tdir_count, v);
        if (v != buf)
                _TIFFfree((char*) v);
        return (status);
}
```

---

kfax'TIFFFetchRefBlackWhite() (./kdegraphics/kfax/libtiffax/tif_dirread.c:1271)

```
TIFFFetchRefBlackWhite(TIFF* tif, TIFFDirEntry* dir)
{
        static char mesg[] = "for \"ReferenceBlackWhite\" array";
        char* cp;
        int ok;

        if (dir->tdir_type == TIFF_RATIONAL)
                return (TIFFFetchNormalTag(tif, dir));
        /*
         * Handle LONG's for backward compatibility.
         */
        cp = CheckMalloc(tif, dir->tdir_count * sizeof (uint32), mesg);
        if (ok = (cp && TIFFFetchLongArray(tif, dir, (uint32*) cp))) {
                float* fp = (float*)
                    CheckMalloc(tif, dir->tdir_count * sizeof (float), mesg);
                if (ok = (fp != NULL)) {
                        uint32 i;
                        for (i = 0; i < dir->tdir_count; i++)
                                fp[i] = (float)((uint32*) cp)[i];
                        ok = TIFFSetField(tif, dir->tdir_tag, fp);
                        _TIFFfree((char*) fp);
                }
        }
        if (cp)
                _TIFFfree(cp);
        return (ok);
}
```

```
#endif

#if STRIPCHOP_SUPPORT
/*
 * Replace a single strip (tile) of uncompressed data by
 * multiple strips (tiles), each approximately 8Kbytes.
 * This is useful for dealing with large images or
 * for dealing with machines with a limited amount
 * memory.
 */
static void
```

---

kfax'ChopUpSingleUncompressedStrip() (./kdegraphics/kfax/libtiffax/tif_dirread.c:1309)

```
ChopUpSingleUncompressedStrip(TIFF* tif)
{
        register TIFFDirectory *td = &tif->tif_dir;
        uint32 bytecount = td->td_stripbytecount[0];
        uint32 offset = td->td_stripoffset[0];
        tsize_t rowbytes = TIFFVTileSize(tif, 1), stripbytes;
        tstrip_t strip, nstrips, rowsperstrip;
        uint32* newcounts;
        uint32* newoffsets;

        /*
         * Make the rows hold at least one
         * scanline, but fill 8k if possible.
         */
        if (rowbytes > 8192) {
                stripbytes = rowbytes;
                rowsperstrip = 1;
        } else {
                rowsperstrip = 8192 / rowbytes;
                stripbytes = rowbytes * rowsperstrip;
        }
        /* never increase the number of strips in an image */
        if (rowsperstrip >= td->td_rowsperstrip)
                return;
        nstrips = (tstrip_t) TIFFhowmany(bytecount, stripbytes);
        newcounts = (uint32*) CheckMalloc(tif, nstrips * sizeof (uint32),
                                "for chopped \"StripByteCounts\" array");
        newoffsets = (uint32*) CheckMalloc(tif, nstrips * sizeof (uint32),
                                "for chopped \"StripOffsets\" array");
        if (newcounts == NULL || newoffsets == NULL) {
                /*
                 * Unable to allocate new strip information, give
                 * up and use the original one strip information.
                 */
                if (newcounts != NULL)
                        _TIFFfree(newcounts);
                if (newoffsets != NULL)
                        _TIFFfree(newoffsets);
                return;
        }
        /*
         * Fill the strip information arrays with
         * new bytecounts and offsets that reflect
```

```
         * the broken-up format.
         */
        for (strip = 0; strip < nstrips; strip++) {
                if (stripbytes > bytecount)
                        stripbytes = bytecount;
                newcounts[strip] = stripbytes;
                newoffsets[strip] = offset;
                offset += stripbytes;
                bytecount -= stripbytes;
        }
        /*
         * Replace old single strip info with multi-strip info.
         */
        td->td_stripsperimage = td->td_nstrips = nstrips;
        TIFFSetField(tif, TIFFTAG_ROWSPERSTRIP, rowsperstrip);

        _TIFFfree(td->td_stripbytecount);
        _TIFFfree(td->td_stripoffset);
        td->td_stripbytecount = newcounts;
        td->td_stripoffset = newoffsets;
}
```

---

kfax'TIFFWriteDirectory() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:88)

```
TIFFWriteDirectory(TIFF* tif)
{
        uint16 dircount;
        uint32 diroff;
        ttag_t tag;
        uint32 nfields;
        tsize_t dirsize;
        char* data;
        TIFFDirEntry* dir;
        TIFFDirectory* td;
        u_long b, fields[FIELD_SETLONGS];
        int fi, nfi;

        if (tif->tif_mode == O_RDONLY)
                return (1);
        /*
         * Clear write state so that subsequent images with
         * different characteristics get the right buffers
         * setup for them.
         */
        if (tif->tif_flags & TIFF_POSTENCODE) {
                tif->tif_flags &= ~TIFF_POSTENCODE;
                if (!(*tif->tif_postencode)(tif)) {
                        TIFFError(tif->tif_name,
                            "Error post-encoding before directory write");
                        return (0);
                }
        }
        (*tif->tif_close)(tif);                 /* shutdown encoder */
        /*
         * Flush any data that might have been written
         * by the compression close+cleanup routines.
         */
        if (tif->tif_rawcc > 0 && !TIFFFlushData1(tif)) {
```

```
                TIFFError(tif->tif_name,
                    "Error flushing data before directory write");
                return (0);
        }
        if ((tif->tif_flags & TIFF_MYBUFFER) && tif->tif_rawdata) {
                _TIFFfree(tif->tif_rawdata);
                tif->tif_rawdata = NULL;
                tif->tif_rawcc = 0;
        }
        tif->tif_flags &= ~(TIFF_BEENWRITING|TIFF_BUFFERSETUP);

        td = &tif->tif_dir;
        /*
         * Size the directory so that we can calculate
         * offsets for the data items that aren't kept
         * in-place in each field.
         */
        nfields = 0;
        for (b = 0; b <= FIELD_LAST; b++)
                if (TIFFFieldSet(tif, b))
                        nfields += (b < FIELD_SUBFILETYPE ? 2 : 1);
        dirsize = nfields * sizeof (TIFFDirEntry);
        data = (char*) _TIFFmalloc(dirsize);
        if (data == NULL) {
                TIFFError(tif->tif_name,
                    "Cannot write directory, out of space");
                return (0);
        }
        /*
         * Directory hasn't been placed yet, put
         * it at the end of the file and link it
         * into the existing directory structure.
         */
        if (tif->tif_diroff == 0 && !TIFFLinkDirectory(tif))
                goto bad;
        tif->tif_dataoff = (toff_t)(
            tif->tif_diroff + sizeof (uint16) + dirsize + sizeof (toff_t));
        if (tif->tif_dataoff & 1)
                tif->tif_dataoff++;
        (void) TIFFSeekFile(tif, tif->tif_dataoff, SEEK_SET);
        tif->tif_curdir++;
        dir = (TIFFDirEntry*) data;
        /*
         * Setup external form of directory
         * entries and write data items.
         */
        _TIFFmemcpy(fields, td->td_fieldsset, sizeof (fields));
        /*
         * Write out ExtraSamples tag only if
         * extra samples are present in the data.
         */
        if (FieldSet(fields, FIELD_EXTRASAMPLES) && !td->td_extrasamples) {
                ResetFieldBit(fields, FIELD_EXTRASAMPLES);
                nfields--;
                dirsize -= sizeof (TIFFDirEntry);
        }                                                               /*XXX*/
        for (fi = 0, nfi = tif->tif_nfields; nfi > 0; nfi--, fi++) {
                const TIFFFieldInfo* fip = tif->tif_fieldinfo[fi];
                if (!FieldSet(fields, fip->field_bit))
                        continue;
                switch (fip->field_bit) {
```

```
        case FIELD_STRIPOFFSETS:
                /*
                 * We use one field bit for both strip and tile
                 * offsets, and so must be careful in selecting
                 * the appropriate field descriptor (so that tags
                 * are written in sorted order).
                 */
                tag = isTiled(tif) ?
                    TIFFTAG_TILEOFFSETS : TIFFTAG_STRIPOFFSETS;
                if (tag != fip->field_tag)
                        continue;
                if (!TIFFWriteLongArray(tif, TIFF_LONG, tag, dir,
                    (uint32) td->td_nstrips, td->td_stripoffset))
                        goto bad;
                break;
        case FIELD_STRIPBYTECOUNTS:
                /*
                 * We use one field bit for both strip and tile
                 * byte counts, and so must be careful in selecting
                 * the appropriate field descriptor (so that tags
                 * are written in sorted order).
                 */
                tag = isTiled(tif) ?
                    TIFFTAG_TILEBYTECOUNTS : TIFFTAG_STRIPBYTECOUNTS;
                if (tag != fip->field_tag)
                        continue;
                if (!TIFFWriteLongArray(tif, TIFF_LONG, tag, dir,
                    (uint32) td->td_nstrips, td->td_stripbytecount))
                        goto bad;
                break;
        case FIELD_ROWSPERSTRIP:
                TIFFSetupShortLong(tif, TIFFTAG_ROWSPERSTRIP,
                    dir, td->td_rowsperstrip);
                break;
        case FIELD_COLORMAP:
                if (!TIFFWriteShortTable(tif, TIFFTAG_COLORMAP, dir,
                    3, td->td_colormap))
                        goto bad;
                break;
        case FIELD_IMAGEDIMENSIONS:
                TIFFSetupShortLong(tif, TIFFTAG_IMAGEWIDTH,
                    dir++, td->td_imagewidth);
                TIFFSetupShortLong(tif, TIFFTAG_IMAGELENGTH,
                    dir, td->td_imagelength);
                break;
        case FIELD_TILEDIMENSIONS:
                TIFFSetupShortLong(tif, TIFFTAG_TILEWIDTH,
                    dir++, td->td_tilewidth);
                TIFFSetupShortLong(tif, TIFFTAG_TILELENGTH,
                    dir, td->td_tilelength);
                break;
        case FIELD_POSITION:
                WriteRationalPair(TIFF_RATIONAL,
                    TIFFTAG_XPOSITION, td->td_xposition,
                    TIFFTAG_YPOSITION, td->td_yposition);
                break;
        case FIELD_RESOLUTION:
                WriteRationalPair(TIFF_RATIONAL,
                    TIFFTAG_XRESOLUTION, td->td_xresolution,
                    TIFFTAG_YRESOLUTION, td->td_yresolution);
                break;
```

```
                case FIELD_BITSPERSAMPLE:
                case FIELD_MINSAMPLEVALUE:
                case FIELD_MAXSAMPLEVALUE:
                case FIELD_SAMPLEFORMAT:
                        if (!TIFFWritePerSampleShorts(tif, fip->field_tag, dir))
                                goto bad;
                        break;
                case FIELD_SMINSAMPLEVALUE:
                case FIELD_SMAXSAMPLEVALUE:
                        if (!TIFFWritePerSampleAnys(tif,
                            _TIFFSampleToTagType(tif), fip->field_tag, dir))
                                goto bad;
                        break;
                case FIELD_PAGENUMBER:
                case FIELD_HALFTONEHINTS:
#ifdef YCBCR_SUPPORT
                case FIELD_YCBCRSUBSAMPLING:
#endif
#ifdef CMYK_SUPPORT
                case FIELD_DOTRANGE:
#endif
                        if (!TIFFSetupShortPair(tif, fip->field_tag, dir))
                                goto bad;
                        break;
#ifdef COLORIMETRY_SUPPORT
                case FIELD_TRANSFERFUNCTION:
                        if (!TIFFWriteTransferFunction(tif, dir))
                                goto bad;
                        break;
#endif
#if SUBIFD_SUPPORT
                case FIELD_SUBIFD:
                        if (!TIFFWriteNormalTag(tif, dir, fip))
                                goto bad;
                        /*
                         * Total hack: if this directory includes a SubIFD
                         * tag then force the next <n> directories to be
                         * written as ``sub directories'' of this one.  This
                         * is used to write things like thumbnails and
                         * image masks that one wants to keep out of the
                         * normal directory linkage access mechanism.
                         */
                        if (dir->tdir_count > 0) {
                                tif->tif_flags |= TIFF_INSUBIFD;
                                tif->tif_nsubifd = dir->tdir_count;
                                if (dir->tdir_count > 1)
                                        tif->tif_subifdoff = dir->tdir_offset;
                                else
                                        tif->tif_subifdoff = (uint32)(
                                              tif->tif_diroff
                                            + sizeof (uint16)
                                            + ((char*)&dir->tdir_offset-data));
                        }
                        break;
#endif
                default:
                        if (!TIFFWriteNormalTag(tif, dir, fip))
                                goto bad;
                        break;
                }
                dir++;
```

```
                ResetFieldBit(fields, fip->field_bit);
        }
        /*
         * Write directory.
         */
        dircount = (uint16) nfields;
        diroff = (uint32) tif->tif_nextdiroff;
        if (tif->tif_flags & TIFF_SWAB) {
                /*
                 * The file's byte order is opposite to the
                 * native machine architecture.  We overwrite
                 * the directory information with impunity
                 * because it'll be released below after we
                 * write it to the file.  Note that all the
                 * other tag construction routines assume that
                 * we do this byte-swapping; i.e. they only
                 * byte-swap indirect data.
                 */
                for (dir = (TIFFDirEntry*) data; dircount; dir++, dircount--) {
                        TIFFSwabArrayOfShort(&dir->tdir_tag, 2);
                        TIFFSwabArrayOfLong(&dir->tdir_count, 2);
                }
                dircount = (uint16) nfields;
                TIFFSwabShort(&dircount);
                TIFFSwabLong(&diroff);
        }
        (void) TIFFSeekFile(tif, tif->tif_diroff, SEEK_SET);
        if (!WriteOK(tif, &dircount, sizeof (dircount))) {
                TIFFError(tif->tif_name, "Error writing directory count");
                goto bad;
        }
        if (!WriteOK(tif, data, dirsize)) {
                TIFFError(tif->tif_name, "Error writing directory contents");
                goto bad;
        }
        if (!WriteOK(tif, &diroff, sizeof (diroff))) {
                TIFFError(tif->tif_name, "Error writing directory link");
                goto bad;
        }
        TIFFFreeDirectory(tif);
        _TIFFfree(data);
        tif->tif_flags &= ~TIFF_DIRTYDIRECT;
        (*tif->tif_cleanup)(tif);

        /*
         * Reset directory-related state for subsequent
         * directories.
         */
        TIFFDefaultDirectory(tif);
        tif->tif_diroff = 0;
        tif->tif_curoff = 0;
        tif->tif_row = (uint32) -1;
        tif->tif_curstrip = (tstrip_t) -1;
        return (1);
bad:
        _TIFFfree(data);
        return (0);
}
```

kfax'TIFFWriteNormalTag() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:369)

```
TIFFWriteNormalTag(TIFF* tif, TIFFDirEntry* dir, const TIFFFieldInfo* fip)
{
        u_short wc = (u_short) fip->field_writecount;

        dir->tdir_tag = fip->field_tag;
        dir->tdir_type = (u_short) fip->field_type;
        dir->tdir_count = wc;
```

---

kfax'TIFFSetupShortLong() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:488)

```
TIFFSetupShortLong(TIFF* tif, ttag_t tag, TIFFDirEntry* dir, uint32 v)
{
        dir->tdir_tag = tag;
        dir->tdir_count = 1;
        if (v > 0xffffL) {
                dir->tdir_type = (short) TIFF_LONG;
                dir->tdir_offset = v;
        } else {
                dir->tdir_type = (short) TIFF_SHORT;
                dir->tdir_offset = TIFFInsertData(tif, (int) TIFF_SHORT, v);
        }
}
```

---

kfax'TIFFWriteRational() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:508)

```
TIFFWriteRational(TIFF* tif,
    TIFFDataType type, ttag_t tag, TIFFDirEntry* dir, float v)
{
        return (TIFFWriteRationalArray(tif, type, tag, dir, 1, &v));
}
```

---

kfax'TIFFWritePerSampleShorts() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:523)

```
TIFFWritePerSampleShorts(TIFF* tif, ttag_t tag, TIFFDirEntry* dir)
{
        uint16 buf[10], v;
        uint16* w = buf;
        int i, status, samples = tif->tif_dir.td_samplesperpixel;

        if (samples > NITEMS(buf))
                w = (uint16*) _TIFFmalloc(samples * sizeof (uint16));
        TIFFGetField(tif, tag, &v);
        for (i = 0; i < samples; i++)
                w[i] = v;
        status = TIFFWriteShortArray(tif, TIFF_SHORT, tag, dir, samples, w);
        if (w != buf)
                _TIFFfree((char*) w);
        return (status);
```

```
}

/*
 * Setup a directory entry that references a samples/pixel array of ``type''
 * values and (potentially) write the associated indirect values.  The source
 * data from TIFFGetField() for the specified tag must be returned as double.
 */
static int
```

---

kfax'TIFFWritePerSampleAnys() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:546)

```
TIFFWritePerSampleAnys(TIFF* tif,
    TIFFDataType type, ttag_t tag, TIFFDirEntry* dir)
{
        double buf[10], v;
        double* w = buf;
        int i, status;
        int samples = (int) tif->tif_dir.td_samplesperpixel;

        if (samples > NITEMS(buf))
                w = (double*) _TIFFmalloc(samples * sizeof (double));
        TIFFGetField(tif, tag, &v);
        for (i = 0; i < samples; i++)
                w[i] = v;
        status = TIFFWriteAnyArray(tif, type, tag, dir, samples, w);
        if (w != buf)
                _TIFFfree(w);
        return (status);
}
```

---

kfax'TIFFSetupShortPair() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:571)

```
TIFFSetupShortPair(TIFF* tif, ttag_t tag, TIFFDirEntry* dir)
{
        uint16 v[2];

        TIFFGetField(tif, tag, &v[0], &v[1]);
        return (TIFFWriteShortArray(tif, TIFF_SHORT, tag, dir, 2, v));
}

/*
 * Setup a directory entry for an NxM table of shorts,
 * where M is known to be 2**bitspersample, and write
 * the associated indirect data.
 */
static int
```

---

kfax'TIFFWriteShortTable() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:585)

```
TIFFWriteShortTable(TIFF* tif,
    ttag_t tag, TIFFDirEntry* dir, uint32 n, uint16** table)
{
```

```
        uint32 i, off;

        dir->tdir_tag = tag;
        dir->tdir_type = (short) TIFF_SHORT;
        /* XXX -- yech, fool TIFFWriteData */
        dir->tdir_count = (uint32) (1L<<tif->tif_dir.td_bitspersample);
        off = tif->tif_dataoff;
        for (i = 0; i < n; i++)
                if (!TIFFWriteData(tif, dir, (char *)table[i]))
                        return (0);
        dir->tdir_count *= n;
        dir->tdir_offset = off;
        return (1);
}

/*
 * Write/copy data associated with an ASCII or opaque tag value.
 */
static int
```

---

kfax'TIFFWriteByteArray() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:607)

```
TIFFWriteByteArray(TIFF* tif, TIFFDirEntry* dir, char* cp)
{
        if (dir->tdir_count > 4) {
                if (!TIFFWriteData(tif, dir, cp))
                        return (0);
        } else
                _TIFFmemcpy(&dir->tdir_offset, cp, dir->tdir_count);
        return (1);
}

/*
 * Setup a directory entry of an array of SHORT
 * or SSHORT and write the associated indirect values.
 */
static int
```

---

kfax'TIFFWriteShortArray() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:622)

```
TIFFWriteShortArray(TIFF* tif,
    TIFFDataType type, ttag_t tag, TIFFDirEntry* dir, uint32 n, uint16* v)
{
        dir->tdir_tag = tag;
        dir->tdir_type = (short) type;
        dir->tdir_count = n;
        if (n <= 2) {
                if (tif->tif_header.tiff_magic == TIFF_BIGENDIAN) {
                        dir->tdir_offset = (uint32) ((long) v[0] << 16);
                        if (n == 2)
                                dir->tdir_offset |= v[1] & 0xffff;
                } else {
                        dir->tdir_offset = v[0] & 0xffff;
                        if (n == 2)
                                dir->tdir_offset |= (long) v[1] << 16;
                }
```

```
                return (1);
        } else
                return (TIFFWriteData(tif, dir, (char*) v));
}

/*
 * Setup a directory entry of an array of LONG
 * or SLONG and write the associated indirect values.
 */
static int
```

---

kfax'TIFFWriteLongArray() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:648)

```
TIFFWriteLongArray(TIFF* tif,
    TIFFDataType type, ttag_t tag, TIFFDirEntry* dir, uint32 n, uint32* v)
{
        dir->tdir_tag = tag;
        dir->tdir_type = (short) type;
        dir->tdir_count = n;
        if (n == 1) {
                dir->tdir_offset = v[0];
                return (1);
        } else
                return (TIFFWriteData(tif, dir, (char*) v));
}

/*
 * Setup a directory entry of an array of RATIONAL
 * or SRATIONAL and write the associated indirect values.
 */
static int
```

---

kfax'TIFFWriteRationalArray() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:666)

```
TIFFWriteRationalArray(TIFF* tif,
    TIFFDataType type, ttag_t tag, TIFFDirEntry* dir, uint32 n, float* v)
{
        uint32 i;
        uint32* t;
        int status;

        dir->tdir_tag = tag;
        dir->tdir_type = (short) type;
        dir->tdir_count = n;
        t = (uint32*) _TIFFmalloc(2*n * sizeof (uint32));
        for (i = 0; i < n; i++) {
                float fv = v[i];
                int sign = 1;
                uint32 den;

                if (fv < 0) {
                        if (type == TIFF_RATIONAL) {
                                TIFFWarning(tif->tif_name,
        "\"%s\": Information lost writing value (%g) as (unsigned) RATIONAL",
                                _TIFFFieldWithTag(tif,tag)->field_name, v);
                                fv = 0;
```

```
                        } else
                                fv = -fv, sign = -1;
                }
                den = 1L;
                if (fv > 0) {
                        while (fv < 1L<<(31-3) && den < 1L<<(31-3))
                                fv *= 1<<3, den *= 1L<<3;
                }
                t[2*i+0] = sign * (fv + 0.5);
                t[2*i+1] = den;
        }
        status = TIFFWriteData(tif, dir, (char *)t);
        _TIFFfree((char*) t);
        return (status);
}

static int
```

---

kfax'TIFFWriteFloatArray() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:705)

```
TIFFWriteFloatArray(TIFF* tif,
    TIFFDataType type, ttag_t tag, TIFFDirEntry* dir, uint32 n, float* v)
{
        dir->tdir_tag = tag;
        dir->tdir_type = (short) type;
        dir->tdir_count = n;
        TIFFCvtNativeToIEEEFloat(tif, n, v);
        if (n == 1) {
                dir->tdir_offset = *(uint32*) &v[0];
                return (1);
        } else
                return (TIFFWriteData(tif, dir, (char*) v));
}

static int
```

---

kfax'TIFFWriteDoubleArray() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:720)

```
TIFFWriteDoubleArray(TIFF* tif,
    TIFFDataType type, ttag_t tag, TIFFDirEntry* dir, uint32 n, double* v)
{
        dir->tdir_tag = tag;
        dir->tdir_type = (short) type;
        dir->tdir_count = n;
        TIFFCvtNativeToIEEEDouble(tif, n, v);
        return (TIFFWriteData(tif, dir, (char*) v));
}

/*
 * Write an array of ``type'' values for a specified tag (i.e. this is a tag
 * which is allowed to have different types, e.g. SMaxSampleType).
 * Internally the data values are represented as double since a double can
 * hold any of the TIFF tag types (yes, this should really be an abstract
 * type tany_t for portability).  The data is converted into the specified
 * type in a temporary buffer and then handed off to the appropriate array
 * writer.
```

```
 */
static int
```

---

kfax'TIFFWriteAnyArray() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:740)

```
TIFFWriteAnyArray(TIFF* tif,
    TIFFDataType type, ttag_t tag, TIFFDirEntry* dir, uint32 n, double* v)
{
        char buf[10 * sizeof(double)];
        char* w = buf;
        int i, status = 0;

        if (n * tiffDataWidth[type] > sizeof buf)
                w = (char*) _TIFFmalloc(n * tiffDataWidth[type]);
        switch (type) {
        case TIFF_BYTE:
                { unsigned char* bp = (unsigned char*) w;
                  for (i = 0; i < n; i++)
                        bp[i] = (unsigned char) v[i];
                  dir->tdir_tag = tag;
                  dir->tdir_type = (short) type;
                  dir->tdir_count = n;
                  if (!TIFFWriteByteArray(tif, dir, (char*) bp))
                        goto out;
                }
                break;
        case TIFF_SBYTE:
                { signed char* bp = (signed char*) w;
                  for (i = 0; i < n; i++)
                        bp[i] = (signed char) v[i];
                  dir->tdir_tag = tag;
                  dir->tdir_type = (short) type;
                  dir->tdir_count = n;
                  if (!TIFFWriteByteArray(tif, dir, (char*) bp))
                        goto out;
                }
                break;
        case TIFF_SHORT:
                { uint16* bp = (uint16*) w;
                  for (i = 0; i < n; i++)
                        bp[i] = (uint16) v[i];
                  if (!TIFFWriteShortArray(tif, type, tag, dir, n, (uint16*)bp))
                                goto out;
                }
                break;
        case TIFF_SSHORT:
                { int16* bp = (int16*) w;
                  for (i = 0; i < n; i++)
                        bp[i] = (int16) v[i];
                  if (!TIFFWriteShortArray(tif, type, tag, dir, n, (uint16*)bp))
                        goto out;
                }
                break;
        case TIFF_LONG:
                { uint32* bp = (uint32*) w;
                  for (i = 0; i < n; i++)
                        bp[i] = (uint32) v[i];
                  if (!TIFFWriteLongArray(tif, type, tag, dir, n, bp))
```

```
                        goto out;
                }
                break;
        case TIFF_SLONG:
                { int32* bp = (int32*) w;
                  for (i = 0; i < n; i++)
                        bp[i] = (int32) v[i];
                  if (!TIFFWriteLongArray(tif, type, tag, dir, n, (uint32*) bp))
                        goto out;
                }
                break;
        case TIFF_FLOAT:
                { float* bp = (float*) w;
                  for (i = 0; i < n; i++)
                        bp[i] = (float) v[i];
                  if (!TIFFWriteFloatArray(tif, type, tag, dir, n, bp))
                        goto out;
                }
                break;
        case TIFF_DOUBLE:
                return (TIFFWriteDoubleArray(tif, type, tag, dir, n, v));
        default:
                /* TIFF_NOTYPE */
                /* TIFF_ASCII */
                /* TIFF_UNDEFINED */
                /* TIFF_RATIONAL */
                /* TIFF_SRATIONAL */
                goto out;
        }
        status = 1;
 out:
        if (w != buf)
                _TIFFfree(w);
        return (status);
}

#ifdef COLORIMETRY_SUPPORT
static int
```

---

kfax'TIFFWriteTransferFunction() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:831)

```
TIFFWriteTransferFunction(TIFF* tif, TIFFDirEntry* dir)
{
        TIFFDirectory* td = &tif->tif_dir;
        tsize_t n = (1L<<td->td_bitspersample) * sizeof (uint16);
        uint16** tf = td->td_transferfunction;
        int ncols;

        /*
         * Check if the table can be written as a single column,
         * or if it must be written as 3 columns.  Note that we
         * write a 3-column tag if there are 2 samples/pixel and
         * a single column of data won't suffice--hmm.
         */
        switch (td->td_samplesperpixel - td->td_extrasamples) {
        default:        if (_TIFFmemcmp(tf[0], tf[2], n)) { ncols = 3; break; }
        case 2:         if (_TIFFmemcmp(tf[0], tf[1], n)) { ncols = 3; break; }
```

```
        case 1: case 0: ncols = 1;
        }
        return (TIFFWriteShortTable(tif,
            TIFFTAG_TRANSFERFUNCTION, dir, ncols, tf));
}
#endif

/*
 * Write a contiguous directory item.
 */
static int
```

---

kfax'TIFFWriteData() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:858)

```
TIFFWriteData(TIFF* tif, TIFFDirEntry* dir, char* cp)
{
        tsize_t cc;

        if (tif->tif_flags & TIFF_SWAB) {
                switch (dir->tdir_type) {
                case TIFF_SHORT:
                case TIFF_SSHORT:
                        TIFFSwabArrayOfShort((uint16*) cp, dir->tdir_count);
                        break;
                case TIFF_LONG:
                case TIFF_SLONG:
                case TIFF_FLOAT:
                        TIFFSwabArrayOfLong((uint32*) cp, dir->tdir_count);
                        break;
                case TIFF_RATIONAL:
                case TIFF_SRATIONAL:
                        TIFFSwabArrayOfLong((uint32*) cp, 2*dir->tdir_count);
                        break;
                case TIFF_DOUBLE:
                        TIFFSwabArrayOfDouble((double*) cp, dir->tdir_count);
                        break;
                }
        }
        dir->tdir_offset = tif->tif_dataoff;
        cc = dir->tdir_count * tiffDataWidth[dir->tdir_type];
        if (SeekOK(tif, dir->tdir_offset) &&
            WriteOK(tif, cp, cc)) {
                tif->tif_dataoff += (cc + 1) & ~1;
                return (1);
        }
        TIFFError(tif->tif_name, "Error writing data for field \"%s\"",
            _TIFFFieldWithTag(tif, dir->tdir_tag)->field_name);
        return (0);
}

/*
 * Link the current directory into the
 * directory chain for the file.
 */
static int
```

---

kfax'TIFFLinkDirectory() (./kdegraphics/kfax/libtiffax/tif_dirwrite.c:899)

```
TIFFLinkDirectory(TIFF* tif)
{
        static const char module[] = "TIFFLinkDirectory";
        uint32 nextdir;
        uint32 diroff;

        tif->tif_diroff = (TIFFSeekFile(tif, (toff_t) 0, SEEK_END)+1) &~ 1;
        diroff = (uint32) tif->tif_diroff;
        if (tif->tif_flags & TIFF_SWAB)
                TIFFSwabLong(&diroff);
#if SUBIFD_SUPPORT
        if (tif->tif_flags & TIFF_INSUBIFD) {
                (void) TIFFSeekFile(tif, tif->tif_subifdoff, SEEK_SET);
                if (!WriteOK(tif, &diroff, sizeof (diroff))) {
                        TIFFError(module,
                            "%s: Error writing SubIFD directory link",
                            tif->tif_name);
                        return (0);
                }
                /*
                 * Advance to the next SubIFD or, if this is
                 * the last one configured, revert back to the
                 * normal directory linkage.
                 */
                if (--tif->tif_nsubifd)
                        tif->tif_subifdoff += sizeof (diroff);
                else
                        tif->tif_flags &= ~TIFF_INSUBIFD;
                return (1);
        }
#endif
        if (tif->tif_header.tiff_diroff == 0) {
                /*
                 * First directory, overwrite offset in header.
                 */
                tif->tif_header.tiff_diroff = diroff;
```

---

kfax'DumpModeEncode() (./kdegraphics/kfax/libtiffax/tif_dumpmode.c:38)

```
DumpModeEncode(TIFF* tif, tidata_t pp, tsize_t cc, tsample_t s)
{
        (void) s;
        while (cc > 0) {
                tsize_t n;

                n = cc;
                if (tif->tif_rawcc + n > tif->tif_rawdatasize)
                        n = tif->tif_rawdatasize - tif->tif_rawcc;
                /*
                 * Avoid copy if client has setup raw
                 * data buffer to avoid extra copy.
                 */
                if (tif->tif_rawcp != pp)
                        _TIFFmemcpy(tif->tif_rawcp, pp, n);
                tif->tif_rawcp += n;
```

```
                tif->tif_rawcc += n;
                pp += n;
                cc -= n;
                if (tif->tif_rawcc >= tif->tif_rawdatasize &&
                    !TIFFFlushData1(tif))
                        return (-1);
        }
        return (1);
}

/*
 * Decode a hunk of pixels.
 */
static int
```

---

kfax'DumpModeDecode() (./kdegraphics/kfax/libtiffax/tif_dumpmode.c:68)

```
DumpModeDecode(TIFF* tif, tidata_t buf, tsize_t cc, tsample_t s)
{
        (void) s;
        if (tif->tif_rawcc < cc) {
                TIFFError(tif->tif_name,
                    "DumpModeDecode: Not enough data for scanline %d",
                    tif->tif_row);
                return (0);
        }
        /*
         * Avoid copy if client has setup raw
         * data buffer to avoid extra copy.
         */
        if (tif->tif_rawcp != buf)
                _TIFFmemcpy(buf, tif->tif_rawcp, cc);
        tif->tif_rawcp += cc;
        tif->tif_rawcc -= cc;
        return (1);
}

/*
 * Seek forwards nrows in the current strip.
 */
static int
```

---

kfax'DumpModeSeek() (./kdegraphics/kfax/libtiffax/tif_dumpmode.c:92)

```
DumpModeSeek(TIFF* tif, uint32 nrows)
{
        tif->tif_rawcp += nrows * tif->tif_scanlinesize;
        tif->tif_rawcc -= nrows * tif->tif_scanlinesize;
        return (1);
}

/*
 * Initialize dump mode.
 */
int
```

kfax'TIFFInitDumpMode() (./kdegraphics/kfax/libtiffax/tif_dumpmode.c:103)

```
TIFFInitDumpMode(TIFF* tif, int scheme)
{
        (void) scheme;
        tif->tif_decoderow = DumpModeDecode;
        tif->tif_decodestrip = DumpModeDecode;
        tif->tif_decodetile = DumpModeDecode;
        tif->tif_encoderow = DumpModeEncode;
        tif->tif_encodestrip = DumpModeEncode;
        tif->tif_encodetile = DumpModeEncode;
        tif->tif_seek = DumpModeSeek;
        return (1);
}
```

kfax'TIFFSetErrorHandler() (./kdegraphics/kfax/libtiffax/tif_error.c:33)

```
TIFFSetErrorHandler(TIFFErrorHandler handler)
{
        TIFFErrorHandler prev = _TIFFerrorHandler;
        _TIFFerrorHandler = handler;
        return (prev);
}

void
```

kfax'TIFFError() (./kdegraphics/kfax/libtiffax/tif_error.c:41)

```
TIFFError(const char* module, const char* fmt, ...)
{
        if (_TIFFerrorHandler) {
                va_list ap;
                va_start(ap, fmt);
                (*_TIFFerrorHandler)(module, fmt, ap);
                va_end(ap);
        }
}
```

kfax'Fax3PreDecode() (./kdegraphics/kfax/libtiffax/tif_fax3.c:151)

```
Fax3PreDecode(TIFF* tif, tsample_t s)
{
        Fax3DecodeState* sp = DecoderState(tif);

        (void) s;
        assert(sp != NULL);
        sp->bit = 0;                    /* force initial read */
        sp->data = 0;
        sp->EOLcnt = 0;                 /* force initial scan for EOL */
        /*
```

```
         * Decoder assumes lsb-to-msb bit order.  Note that we select
         * this here rather than in Fax3SetupState so that viewers can
         * hold the image open, fiddle with the FillOrder tag value,
         * and then re-decode the image.  Otherwise they'd need to close
         * and open the image to get the state reset.
         */
        sp->bitmap =
            TIFFGetBitRevTable(tif->tif_dir.td_fillorder != FILLORDER_LSB2MSB);
        if (sp->refruns) {              /* init reference line to white */
                sp->refruns[0] = sp->b.rowpixels;
                sp->refruns[1] = 0;
        }
        return (1);
}

/*
 * Routine for handling various errors/conditions.
 * Note how they are "glued into the decoder" by
 * overriding the definitions used by the decoder.
 */

static void
```

---

kfax'Fax3Unexpected() (./kdegraphics/kfax/libtiffax/tif_fax3.c:183)

```
Fax3Unexpected(const char* module, TIFF* tif, uint32 a0)
{
        TIFFError(module, "%s: Bad code word at scanline %d (x %lu)",
            tif->tif_name, tif->tif_row, (u_long) a0);
}
```

---

kfax'Fax3Extension() (./kdegraphics/kfax/libtiffax/tif_fax3.c:191)

```
Fax3Extension(const char* module, TIFF* tif, uint32 a0)
{
        TIFFError(module,
            "%s: Uncompressed data (not supported) at scanline %d (x %lu)",
            tif->tif_name, tif->tif_row, (u_long) a0);
}
```

---

kfax'Fax3BadLength() (./kdegraphics/kfax/libtiffax/tif_fax3.c:200)

```
Fax3BadLength(const char* module, TIFF* tif, uint32 a0, uint32 lastx)
{
        TIFFWarning(module, "%s: %s at scanline %d (got %lu, expected %lu)",
            tif->tif_name,
            a0 < lastx ? "Premature EOL" : "Line length mismatch",
            tif->tif_row, (u_long) a0, (u_long) lastx);
}
```

---

kfax'Fax3PrematureEOF() (./kdegraphics/kfax/libtiffax/tif_fax3.c:210)

```
Fax3PrematureEOF(const char* module, TIFF* tif, uint32 a0)
{
        TIFFWarning(module, "%s: Premature EOF at scanline %d (x %lu)",
            tif->tif_name, tif->tif_row, (u_long) a0);
}
```

---

kfax'Fax3Decode1D() (./kdegraphics/kfax/libtiffax/tif_fax3.c:223)

```
Fax3Decode1D(TIFF* tif, tidata_t buf, tsize_t occ, tsample_t s)
{
        DECLARE_STATE(tif, sp, "Fax3Decode1D");

        (void) s;
        CACHE_STATE(tif, sp);
        thisrun = sp->curruns;
        while ((long)occ > 0) {
                a0 = 0;
                RunLength = 0;
                pa = thisrun;
#ifdef FAX3_DEBUG
                printf("\nBitAcc=%08X, BitsAvail = %d\n", BitAcc, BitsAvail);
                printf("-------------------- %d\n", tif->tif_row);
                fflush(stdout);
#endif
                SYNC_EOL(EOF1D);
                EXPAND1D(EOF1Da);
                (*sp->fill)(buf, thisrun, pa, lastx);
                buf += sp->b.rowbytes;
                occ -= sp->b.rowbytes;
                if (occ != 0)
                        tif->tif_row++;
                continue;
        EOF1D:                          /* premature EOF */
                CLEANUP_RUNS();
        EOF1Da:                         /* premature EOF */
                (*sp->fill)(buf, thisrun, pa, lastx);
                UNCACHE_STATE(tif, sp);
                return (-1);
        }
        UNCACHE_STATE(tif, sp);
        return (1);
}
```

---

kfax'Fax3Decode2D() (./kdegraphics/kfax/libtiffax/tif_fax3.c:263)

```
Fax3Decode2D(TIFF* tif, tidata_t buf, tsize_t occ, tsample_t s)
{
        DECLARE_STATE_2D(tif, sp, "Fax3Decode2D");
        int is1D;                       /* current line is 1d/2d-encoded */

        (void) s;
        CACHE_STATE(tif, sp);
        while ((long)occ > 0) {
                a0 = 0;
```

```
                RunLength = 0;
                pa = thisrun = sp->curruns;
#ifdef FAX3_DEBUG
                printf("\nBitAcc=%08X, BitsAvail = %d EOLcnt = %d",
                    BitAcc, BitsAvail, EOLcnt);
#endif
                SYNC_EOL(EOF2D);
                NeedBits8(1, EOF2D);
                is1D = GetBits(1);      /* 1D/2D-encoding tag bit */
                ClrBits(1);
#ifdef FAX3_DEBUG
                printf(" %s\n-------------------- %d\n",
                    is1D ? "1D" : "2D", tif->tif_row);
                fflush(stdout);
#endif
                pb = sp->refruns;
                b1 = *pb++;
                if (is1D)
                        EXPAND1D(EOF2Da);
                else
                        EXPAND2D(EOF2Da);
                (*sp->fill)(buf, thisrun, pa, lastx);
                SETVAL(0);              /* imaginary change for reference */
                SWAP(uint16*, sp->curruns, sp->refruns);
                buf += sp->b.rowbytes;
                occ -= sp->b.rowbytes;
                if (occ != 0)
                        tif->tif_row++;
                continue;
        EOF2D:                          /* premature EOF */
                CLEANUP_RUNS();
        EOF2Da:                         /* premature EOF */
                (*sp->fill)(buf, thisrun, pa, lastx);
                UNCACHE_STATE(tif, sp);
                return (-1);
        }
        UNCACHE_STATE(tif, sp);
        return (1);
}
```

---

kfax'_TIFFFax3fillruns() (./kdegraphics/kfax/libtiffax/tif_fax3.c:358)

```
_TIFFFax3fillruns(u_char* buf, uint16* runs, uint16* erun, uint32 lastx)
{
        static const unsigned char _fillmasks[] =
            { 0x00, 0x80, 0xc0, 0xe0, 0xf0, 0xf8, 0xfc, 0xfe, 0xff };
        u_char* cp;
        uint32 x, bx, run;
        int32 n, nw;
        long* lp;

        if ((erun-runs)&1)
            *erun++ = 0;
        x = 0;
        for (; runs < erun; runs += 2) {
            run = runs[0];
            if (x+run > lastx)
                run = runs[0] = lastx - x;
```

```
            if (run) {
                cp = buf + (x>>3);
                bx = x&7;
                if (run > 8-bx) {
                    if (bx) {                   /* align to byte boundary */
                        *cp++ &= 0xff << (8-bx);
                        run -= 8-bx;
                    }
                    if (n = run >> 3) {         /* multiple bytes to fill */
                        if ((n/sizeof (long)) > 1) {
                            /*
                             * Align to longword boundary and fill.
                             */
                            for (; n && !isAligned(cp, long); n--)
                                    *cp++ = 0x00;
                            lp = (long*) cp;
                            nw = (int32)(n / sizeof (long));
                            n -= nw * sizeof (long);
                            do {
                                    *lp++ = 0L;
                            } while (--nw);
                            cp = (u_char*) lp;
                        }
                        ZERO(n, cp);
                        run &= 7;
                    }
#ifdef PURIFY
                    if (run)
                        cp[0] &= 0xff >> run;
#else
                    cp[0] &= 0xff >> run;
#endif
                } else
                    cp[0] &= ~(_fillmasks[run]>>bx);
                x += runs[0];
            }
            run = runs[1];
            if (x+run > lastx)
                run = runs[1] = lastx - x;
            if (run) {
                cp = buf + (x>>3);
                bx = x&7;
                if (run > 8-bx) {
                    if (bx) {                   /* align to byte boundary */
                        *cp++ |= 0xff >> bx;
                        run -= 8-bx;
                    }
                    if (n = run>>3) {           /* multiple bytes to fill */
                        if ((n/sizeof (long)) > 1) {
                            /*
                             * Align to longword boundary and fill.
                             */
                            for (; n && !isAligned(cp, long); n--)
                                *cp++ = 0xff;
                            lp = (long*) cp;
                            nw = (int32)(n / sizeof (long));
                            n -= nw * sizeof (long);
                            do {
                                *lp++ = -1L;
                            } while (--nw);
                            cp = (u_char*) lp;
```

```
                        }
                        FILL(n, cp);
                        run &= 7;
                    }
#ifdef PURIFY
                    if (run)
                        cp[0] |= 0xff00 >> run;
#else
                    cp[0] |= 0xff00 >> run;
#endif
                } else
                    cp[0] |= _fillmasks[run]>>bx;
                x += runs[1];
            }
        }
        assert(x == lastx);
}
```

---

kfax'Fax3SetupState() (./kdegraphics/kfax/libtiffax/tif_fax3.c:463)

```
Fax3SetupState(TIFF* tif)
{
        TIFFDirectory* td = &tif->tif_dir;
        Fax3BaseState* sp = Fax3State(tif);
        long rowbytes, rowpixels;
        int needsRefLine;

        if (td->td_bitspersample != 1) {
                TIFFError(tif->tif_name,
                    "Bits/sample must be 1 for Group 3/4 encoding/decoding");
                return (0);
        }
        /*
         * Calculate the scanline/tile widths.
         */
        if (isTiled(tif)) {
                rowbytes = TIFFTileRowSize(tif);
                rowpixels = td->td_tilewidth;
        } else {
                rowbytes = TIFFScanlineSize(tif);
                rowpixels = td->td_imagewidth;
        }
        sp->rowbytes = (uint32) rowbytes;
        sp->rowpixels = (uint32) rowpixels;
        /*
         * Allocate any additional space required for decoding/encoding.
         */
        needsRefLine = (
            (sp->groupoptions & GROUP3OPT_2DENCODING) ||
            td->td_compression == COMPRESSION_CCITTFAX4
        );
        if (tif->tif_mode == O_RDONLY) {        /* 1d/2d decoding */
                Fax3DecodeState* dsp = DecoderState(tif);
                uint32 nruns = needsRefLine ?
                     2*TIFFroundup(rowpixels,32) : rowpixels;

                dsp->runs = (uint16*) _TIFFmalloc(nruns*sizeof (uint16));
                if (dsp->runs == NULL) {
```

```
                        TIFFError("Fax3SetupState",
                            "%s: No space for Group 3/4 run arrays",
                            tif->tif_name);
                        return (0);
                }
                dsp->curruns = dsp->runs;
                if (needsRefLine)
                        dsp->refruns = dsp->runs + (nruns>>1);
                else
                        dsp->refruns = NULL;
                if (is2DEncoding(dsp)) {        /* NB: default is 1D routine */
                        tif->tif_decoderow = Fax3Decode2D;
                        tif->tif_decodestrip = Fax3Decode2D;
                        tif->tif_decodetile = Fax3Decode2D;
                }
        } else if (needsRefLine) {              /* 2d encoding */
                Fax3EncodeState* esp = EncoderState(tif);
                /*
                 * 2d encoding requires a scanline
                 * buffer for the ``reference line''; the
                 * scanline against which delta encoding
                 * is referenced.  The reference line must
                 * be initialized to be ``white'' (done elsewhere).
                 */
                esp->refline = (u_char*) _TIFFmalloc(rowbytes);
                if (esp->refline == NULL) {
                        TIFFError("Fax3SetupState",
                            "%s: No space for Group 3/4 reference line",
                            tif->tif_name);
                        return (0);
                }
        } else                                  /* 1d encoding */
                EncoderState(tif)->refline = NULL;
        return (1);
}

/*
 * CCITT Group 3 FAX Encoding.
 */
```

---

kfax'Fax3PutBits() (./kdegraphics/kfax/libtiffax/tif_fax3.c:575)

```
Fax3PutBits(TIFF* tif, u_int bits, u_int length)
{
        Fax3EncodeState* sp = EncoderState(tif);
        int bit = sp->bit;
        int data = sp->data;

        _PutBits(tif, bits, length);

        sp->data = data;
        sp->bit = bit;
}

/*
 * Write a code to the output stream.
 */
```

kfax'putspan() (./kdegraphics/kfax/libtiffax/tif_fax3.c:599)

```
putspan(TIFF* tif, int32 span, const tableentry* tab)
{
        Fax3EncodeState* sp = EncoderState(tif);
        int bit = sp->bit;
        int data = sp->data;
        u_int code, length;

        while (span >= 2624) {
                const tableentry* te = &tab[63 + (2560>>6)];
                code = te->code, length = te->length;
                _PutBits(tif, code, length);
                span -= te->runlen;
        }
        if (span >= 64) {
                const tableentry* te = &tab[63 + (span>>6)];
                assert(te->runlen == 64*(span>>6));
                code = te->code, length = te->length;
                _PutBits(tif, code, length);
                span -= te->runlen;
        }
        code = tab[span].code, length = tab[span].length;
        _PutBits(tif, code, length);

        sp->data = data;
        sp->bit = bit;
}

/*
 * Write an EOL code to the output stream.  The zero-fill
 * logic for byte-aligning encoded scanlines is handled
 * here.  We also handle writing the tag bit for the next
 * scanline when doing 2d encoding.
 */
static void
```

kfax'Fax3PutEOL() (./kdegraphics/kfax/libtiffax/tif_fax3.c:633)

```
Fax3PutEOL(TIFF* tif)
{
        Fax3EncodeState* sp = EncoderState(tif);
        int bit = sp->bit;
        int data = sp->data;
        u_int code, length;

        if (sp->b.groupoptions & GROUP3OPT_FILLBITS) {
                /*
                 * Force bit alignment so EOL will terminate on
                 * a byte boundary.  That is, force the bit alignment
                 * to 16-12 = 4 before putting out the EOL code.
                 */
                int align = 8 - 4;
                if (align != sp->bit) {
                        if (align > sp->bit)
```

```
                                align = sp->bit + (8 - align);
                        else
                                align = sp->bit - align;
                        code = 0;
                        _PutBits(tif, 0, align);
                }
        }
        code = EOL, length = 12;
        if (is2DEncoding(sp))
                code = (code<<1) | (sp->tag == G3_1D), length++;
        _PutBits(tif, code, length);

        sp->data = data;
        sp->bit = bit;
}

/*
 * Reset encoding state at the start of a strip.
 */
static int
```

---

kfax'Fax3PreEncode() (./kdegraphics/kfax/libtiffax/tif_fax3.c:669)

```
Fax3PreEncode(TIFF* tif, tsample_t s)
{
        Fax3EncodeState* sp = EncoderState(tif);

        (void) s;
        assert(sp != NULL);
        sp->bit = 8;
        sp->data = 0;
        sp->tag = G3_1D;
        /*
         * This is necessary for Group 4; otherwise it isn't
         * needed because the first scanline of each strip ends
         * up being copied into the refline.
         */
        if (sp->refline)
                _TIFFmemset(sp->refline, 0x00, sp->b.rowbytes);
        if (is2DEncoding(sp)) {
                float res = tif->tif_dir.td_yresolution;
                /*
                 * The CCITT spec says that when doing 2d encoding, you
                 * should only do it on K consecutive scanlines, where K
                 * depends on the resolution of the image being encoded
                 * (2 for <= 200 lpi, 4 for > 200 lpi).  Since the directory
                 * code initializes td_yresolution to 0, this code will
                 * select a K of 2 unless the YResolution tag is set
                 * appropriately.  (Note also that we fudge a little here
                 * and use 150 lpi to avoid problems with units conversion.)
                 */
                if (tif->tif_dir.td_resolutionunit == RESUNIT_CENTIMETER)
                        res = (res * .3937f) / 2.54f;   /* convert to inches */
                sp->maxk = (res > 150 ? 4 : 2);
                sp->k = sp->maxk-1;
        } else
                sp->k = sp->maxk = 0;
        return (1);
```

```
}
```

---

kfax'find0span() (./kdegraphics/kfax/libtiffax/tif_fax3.c:759)

```
find0span(u_char* bp, int32 bs, int32 be)
{
        int32 bits = be - bs;
        int32 n, span;

        bp += bs>>3;
        /*
         * Check partial byte on lhs.
         */
        if (bits > 0 && (n = (bs & 7))) {
                span = zeroruns[(*bp << n) & 0xff];
                if (span > 8-n)         /* table value too generous */
                        span = 8-n;
                if (span > bits)        /* constrain span to bit range */
                        span = bits;
                if (n+span < 8)         /* doesn't extend to edge of byte */
                        return (span);
                bits -= span;
                bp++;
        } else
                span = 0;
        if (bits >= 2*8*sizeof (long)) {
                long* lp;
                /*
                 * Align to longword boundary and check longwords.
                 */
                while (!isAligned(bp, long)) {
                        if (*bp != 0x00)
                                return (span + zeroruns[*bp]);
                        span += 8, bits -= 8;
                        bp++;
                }
                lp = (long*) bp;
                while (bits >= 8*sizeof (long) && *lp == 0) {
                        span += 8*sizeof (long), bits -= 8*sizeof (long);
                        lp++;
                }
                bp = (u_char*) lp;
        }
        /*
         * Scan full bytes for all 0's.
         */
        while (bits >= 8) {
                if (*bp != 0x00)        /* end of run */
                        return (span + zeroruns[*bp]);
                span += 8, bits -= 8;
                bp++;
        }
        /*
         * Check partial byte on rhs.
         */
        if (bits > 0) {
                n = zeroruns[*bp];
```

```
                span += (n > bits ? bits : n);
        }
        return (span);
}

INLINE static int32
```

---

kfax'find1span() (./kdegraphics/kfax/libtiffax/tif_fax3.c:818)

```
find1span(u_char* bp, int32 bs, int32 be)
{
        int32 bits = be - bs;
        int32 n, span;

        bp += bs>>3;
        /*
         * Check partial byte on lhs.
         */
        if (bits > 0 && (n = (bs & 7))) {
                span = oneruns[(*bp << n) & 0xff];
                if (span > 8-n)         /* table value too generous */
                        span = 8-n;
                if (span > bits)        /* constrain span to bit range */
                        span = bits;
                if (n+span < 8)         /* doesn't extend to edge of byte */
                        return (span);
                bits -= span;
                bp++;
        } else
                span = 0;
        if (bits >= 2*8*sizeof (long)) {
                long* lp;
                /*
                 * Align to longword boundary and check longwords.
                 */
                while (!isAligned(bp, long)) {
                        if (*bp != 0xff)
                                return (span + oneruns[*bp]);
                        span += 8, bits -= 8;
                        bp++;
                }
                lp = (long*) bp;
                while (bits >= 8*sizeof (long) && *lp == ~0) {
                        span += 8*sizeof (long), bits -= 8*sizeof (long);
                        lp++;
                }
                bp = (u_char*) lp;
        }
        /*
         * Scan full bytes for all 1's.
         */
        while (bits >= 8) {
                if (*bp != 0xff)        /* end of run */
                        return (span + oneruns[*bp]);
                span += 8, bits -= 8;
                bp++;
        }
        /*
```

```
         * Check partial byte on rhs.
         */
        if (bits > 0) {
                n = oneruns[*bp];
                span += (n > bits ? bits : n);
        }
        return (span);
}

/*
 * Return the offset of the next bit in the range
 * [bs..be] that is different from the specified
 * color.  The end, be, is returned if no such bit
 * exists.
 */
```

---

kfax'Fax3Encode1DRow() (./kdegraphics/kfax/libtiffax/tif_fax3.c:897)

```
Fax3Encode1DRow(TIFF* tif, u_char* bp, uint32 bits)
{
        Fax3EncodeState* sp = EncoderState(tif);
        int32 bs = 0, span;

        for (;;) {
                span = find0span(bp, bs, bits);         /* white span */
                putspan(tif, span, TIFFFaxWhiteCodes);
                bs += span;
                if (bs >= bits)
                        break;
                span = find1span(bp, bs, bits);         /* black span */
                putspan(tif, span, TIFFFaxBlackCodes);
                bs += span;
                if (bs >= bits)
                        break;
        }
        if (sp->b.mode & (FAXMODE_BYTEALIGN|FAXMODE_WORDALIGN)) {
                if (sp->bit != 8)                       /* byte-align */
                        Fax3FlushBits(tif, sp);
                if ((sp->b.mode&FAXMODE_WORDALIGN) &&
                    !isAligned(tif->tif_rawcp, uint16))
                        Fax3FlushBits(tif, sp);
        }
        return (1);
}
```

---

kfax'Fax3Encode2DRow() (./kdegraphics/kfax/libtiffax/tif_fax3.c:943)

```
Fax3Encode2DRow(TIFF* tif, u_char* bp, u_char* rp, uint32 bits)
{
```

---

kfax'Fax3Encode() (./kdegraphics/kfax/libtiffax/tif_fax3.c:988)

```
Fax3Encode(TIFF* tif, tidata_t bp, tsize_t cc, tsample_t s)
{
        Fax3EncodeState* sp = EncoderState(tif);

        (void) s;
        while ((long)cc > 0) {
                if ((sp->b.mode & FAXMODE_NOEOL) == 0)
                        Fax3PutEOL(tif);
                if (is2DEncoding(sp)) {
                        if (sp->tag == G3_1D) {
                                if (!Fax3Encode1DRow(tif, bp, sp->b.rowpixels))
                                        return (0);
                                sp->tag = G3_2D;
                        } else {
                                if (!Fax3Encode2DRow(tif, bp, sp->refline, sp->b
                                        return (0);
                                sp->k--;
                        }
                        if (sp->k == 0) {
                                sp->tag = G3_1D;
                                sp->k = sp->maxk-1;
                        } else
                                _TIFFmemcpy(sp->refline, bp, sp->b.rowbytes);
                } else {
                        if (!Fax3Encode1DRow(tif, bp, sp->b.rowpixels))
                                return (0);
                }
                bp += sp->b.rowbytes;
                cc -= sp->b.rowbytes;
                if (cc != 0)
                        tif->tif_row++;
        }
        return (1);
}

static int
```

---

kfax'Fax3PostEncode() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1024)

```
Fax3PostEncode(TIFF* tif)
{
        Fax3EncodeState* sp = EncoderState(tif);

        if (sp->bit != 8)
                Fax3FlushBits(tif, sp);
        return (1);
}

static void
```

---

kfax'Fax3Close() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1034)

```
Fax3Close(TIFF* tif)
{
        if ((Fax3State(tif)->mode & FAXMODE_NORTC) == 0) {
```

```
                Fax3EncodeState* sp = EncoderState(tif);
                u_int code = EOL;
                u_int length = 12;
                int i;

                if (is2DEncoding(sp))
                        code = (code<<1) | (sp->tag == G3_1D), length++;
                for (i = 0; i < 6; i++)
                        Fax3PutBits(tif, code, length);
                Fax3FlushBits(tif, sp);
        }
}

static void
```

---

kfax'Fax3Cleanup() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1051)

```
Fax3Cleanup(TIFF* tif)
{
        if (tif->tif_data) {
                if (tif->tif_mode == O_RDONLY) {
                        Fax3DecodeState* sp = DecoderState(tif);
                        if (sp->runs)
                                _TIFFfree(sp->runs);
                } else {
                        Fax3EncodeState* sp = EncoderState(tif);
                        if (sp->refline)
                                _TIFFfree(sp->refline);
                }
                _TIFFfree(tif->tif_data);
                tif->tif_data = NULL;
        }
}
```

---

kfax'Fax3VSetField() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1109)

```
Fax3VSetField(TIFF* tif, ttag_t tag, va_list ap)
{
        Fax3BaseState* sp = Fax3State(tif);

        switch (tag) {
        case TIFFTAG_FAXMODE:
                sp->mode = va_arg(ap, int);
                return (1);                     /* NB: pseudo tag */
        case TIFFTAG_FAXFILLFUNC:
                if (tif->tif_mode == O_RDONLY)
                        DecoderState(tif)->fill = va_arg(ap, TIFFFaxFillFunc);
                return (1);                     /* NB: pseudo tag */
        case TIFFTAG_GROUP3OPTIONS:
        case TIFFTAG_GROUP4OPTIONS:
                sp->groupoptions = va_arg(ap, uint32);
                break;
        case TIFFTAG_BADFAXLINES:
                sp->badfaxlines = va_arg(ap, uint32);
                break;
```

```
        case TIFFTAG_CLEANFAXDATA:
                sp->cleanfaxdata = (uint16) va_arg(ap, int);
                break;
        case TIFFTAG_CONSECUTIVEBADFAXLINES:
                sp->badfaxrun = va_arg(ap, uint32);
                break;
        default:
                return (*sp->vsetparent)(tif, tag, ap);
        }
        TIFFSetFieldBit(tif, _TIFFFieldWithTag(tif, tag)->field_bit);
        tif->tif_flags |= TIFF_DIRTYDIRECT;
        return (1);
}

static int
```

---

kfax'Fax3VGetField() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1143)

```
Fax3VGetField(TIFF* tif, ttag_t tag, va_list ap)
{
        Fax3BaseState* sp = Fax3State(tif);

        switch (tag) {
        case TIFFTAG_FAXMODE:
                *va_arg(ap, int*) = sp->mode;
                break;
        case TIFFTAG_FAXFILLFUNC:
                if (tif->tif_mode == O_RDONLY)
                        *va_arg(ap, TIFFFaxFillFunc*) = DecoderState(tif)->fill;
                break;
        case TIFFTAG_GROUP3OPTIONS:
        case TIFFTAG_GROUP4OPTIONS:
                *va_arg(ap, uint32*) = sp->groupoptions;
                break;
        case TIFFTAG_BADFAXLINES:
                *va_arg(ap, uint32*) = sp->badfaxlines;
                break;
        case TIFFTAG_CLEANFAXDATA:
                *va_arg(ap, uint16*) = sp->cleanfaxdata;
                break;
        case TIFFTAG_CONSECUTIVEBADFAXLINES:
                *va_arg(ap, uint32*) = sp->badfaxrun;
                break;
        default:
                return (*sp->vgetparent)(tif, tag, ap);
        }
        return (1);
}

static void
```

---

kfax'Fax3PrintDir() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1175)

```
Fax3PrintDir(TIFF* tif, FILE* fd, long flags)
{
        Fax3BaseState* sp = Fax3State(tif);
```

```
        (void) flags;
        if (TIFFFieldSet(tif,FIELD_OPTIONS)) {
                const char* sep = " ";
                if (tif->tif_dir.td_compression == COMPRESSION_CCITTFAX4) {
                        fprintf(fd, "  Group 4 Options:");
                        if (sp->groupoptions & GROUP4OPT_UNCOMPRESSED)
                                fprintf(fd, "%suncompressed data", sep);
                } else {

                        fprintf(fd, "  Group 3 Options:");
                        if (sp->groupoptions & GROUP3OPT_2DENCODING)
                                fprintf(fd, "%s2-d encoding", sep), sep = "+";
                        if (sp->groupoptions & GROUP3OPT_FILLBITS)
                                fprintf(fd, "%sEOL padding", sep), sep = "+";
                        if (sp->groupoptions & GROUP3OPT_UNCOMPRESSED)
                                fprintf(fd, "%suncompressed data", sep);
                }
                fprintf(fd, " (%lu = 0x%lx)\n",
                    (u_long) sp->groupoptions, (u_long) sp->groupoptions);
        }
        if (TIFFFieldSet(tif,FIELD_CLEANFAXDATA)) {
                fprintf(fd, "  Fax Data:");
                switch (sp->cleanfaxdata) {
                case CLEANFAXDATA_CLEAN:
                        fprintf(fd, " clean");
                        break;
                case CLEANFAXDATA_REGENERATED:
                        fprintf(fd, " receiver regenerated");
                        break;
                case CLEANFAXDATA_UNCLEAN:
                        fprintf(fd, " uncorrected errors");
                        break;
                }
                fprintf(fd, " (%u = 0x%x)\n",
                    sp->cleanfaxdata, sp->cleanfaxdata);
        }
        if (TIFFFieldSet(tif,FIELD_BADFAXLINES))
                fprintf(fd, "  Bad Fax Lines: %lu\n", (u_long) sp->badfaxlines);
        if (TIFFFieldSet(tif,FIELD_BADFAXRUN))
                fprintf(fd, "  Consecutive Bad Fax Lines: %lu\n",
                    (u_long) sp->badfaxrun);
}

int
```

---

kfax'TIFFInitCCITTFax3() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1223)

```
TIFFInitCCITTFax3(TIFF* tif, int scheme)
{
        Fax3BaseState* sp;

        /*
         * Allocate state block so tag methods have storage to record values.
         */
        if (tif->tif_mode == O_RDONLY)
                tif->tif_data = _TIFFmalloc(sizeof (Fax3DecodeState));
        else
```

```
		tif->tif_data = _TIFFmalloc(sizeof (Fax3EncodeState));
	if (tif->tif_data == NULL) {
		TIFFError("TIFFInitCCITTFax3",
		    "%s: No space for state block", tif->tif_name);
		return (0);
	}
	sp = Fax3State(tif);

	/*
	 * Merge codec-specific tag information and
	 * override parent get/set field methods.
	 */
	switch (scheme) {
	case COMPRESSION_CCITTFAX3:
		_TIFFMergeFieldInfo(tif, fax3FieldInfo, N(fax3FieldInfo));
		break;
	case COMPRESSION_CCITTFAX4:
		_TIFFMergeFieldInfo(tif, fax4FieldInfo, N(fax4FieldInfo));
		break;
	}
	sp->vgetparent = tif->tif_vgetfield;
	tif->tif_vgetfield = Fax3VGetField;	/* hook for codec tags */
	sp->vsetparent = tif->tif_vsetfield;
	tif->tif_vsetfield = Fax3VSetField;	/* hook for codec tags */
	tif->tif_printdir = Fax3PrintDir;	/* hook for codec tags */
	sp->groupoptions = 0;

	TIFFSetField(tif, TIFFTAG_FAXMODE, FAXMODE_CLASSF);
	if (tif->tif_mode == O_RDONLY) {
		tif->tif_flags |= TIFF_NOBITREV;/* decoder does bit reversal */
		DecoderState(tif)->runs = NULL;
		TIFFSetField(tif, TIFFTAG_FAXFILLFUNC, _TIFFFax3fillruns);
	} else
		EncoderState(tif)->refline = NULL;

	/*
	 * Install codec methods.
	 */
	tif->tif_setupdecode = Fax3SetupState;
	tif->tif_predecode = Fax3PreDecode;
	tif->tif_decoderow = Fax3Decode1D;
	tif->tif_decodestrip = Fax3Decode1D;
	tif->tif_decodetile = Fax3Decode1D;
	tif->tif_setupencode = Fax3SetupState;
	tif->tif_preencode = Fax3PreEncode;
	tif->tif_postencode = Fax3PostEncode;
	tif->tif_encoderow = Fax3Encode;
	tif->tif_encodestrip = Fax3Encode;
	tif->tif_encodetile = Fax3Encode;
	tif->tif_close = Fax3Close;
	tif->tif_cleanup = Fax3Cleanup;

	return (1);
}

/*
 * CCITT Group 4 (T.6) Facsimile-compatible
 * Compression Scheme Support.
 */
```

kfax'Fax4Decode() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1298)

```
Fax4Decode(TIFF* tif, tidata_t buf, tsize_t occ, tsample_t s)
{
        DECLARE_STATE_2D(tif, sp, "Fax4Decode");

        (void) s;
        CACHE_STATE(tif, sp);
        while ((long)occ > 0) {
                a0 = 0;
                RunLength = 0;
                pa = thisrun = sp->curruns;
                pb = sp->refruns;
                b1 = *pb++;
#ifdef FAX3_DEBUG
                printf("\nBitAcc=%08X, BitsAvail = %d\n", BitAcc, BitsAvail);
                printf("-------------------- %d\n", tif->tif_row);
                fflush(stdout);
#endif
                EXPAND2D(EOFG4);
                (*sp->fill)(buf, thisrun, pa, lastx);
                SETVAL(0);              /* imaginary change for reference */
                SWAP(uint16*, sp->curruns, sp->refruns);
                buf += sp->b.rowbytes;
                occ -= sp->b.rowbytes;
                if (occ != 0)
                        tif->tif_row++;
                continue;
        EOFG4:
                (*sp->fill)(buf, thisrun, pa, lastx);
                UNCACHE_STATE(tif, sp);
                return (-1);
        }
        UNCACHE_STATE(tif, sp);
        return (1);
}
```

kfax'Fax4Encode() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1338)

```
Fax4Encode(TIFF* tif, tidata_t bp, tsize_t cc, tsample_t s)
{
        Fax3EncodeState *sp = EncoderState(tif);

        (void) s;
        while ((long)cc > 0) {
                if (!Fax3Encode2DRow(tif, bp, sp->refline, sp->b.rowpixels))
                        return (0);
                _TIFFmemcpy(sp->refline, bp, sp->b.rowbytes);
                bp += sp->b.rowbytes;
                cc -= sp->b.rowbytes;
                if (cc != 0)
                        tif->tif_row++;
        }
        return (1);
}
```

```
static int
```

---

kfax'Fax4PostEncode() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1356)

```
Fax4PostEncode(TIFF* tif)
{
        Fax3EncodeState *sp = EncoderState(tif);

        /* terminate strip w/ EOFB */
        Fax3PutBits(tif, EOL, 12);
        Fax3PutBits(tif, EOL, 12);
        if (sp->bit != 8)
                Fax3FlushBits(tif, sp);
        return (1);
}

int
```

---

kfax'TIFFInitCCITTFax4() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1369)

```
TIFFInitCCITTFax4(TIFF* tif, int scheme)
{
        if (TIFFInitCCITTFax3(tif, scheme)) {   /* reuse G3 logic */
                tif->tif_decoderow = Fax4Decode;
                tif->tif_decodestrip = Fax4Decode;
                tif->tif_decodetile = Fax4Decode;
                tif->tif_encoderow = Fax4Encode;
                tif->tif_encodestrip = Fax4Encode;
                tif->tif_encodetile = Fax4Encode;
                tif->tif_postencode = Fax4PostEncode;
                /*
                 * Suppress RTC at the end of each strip.
                 */
                return TIFFSetField(tif, TIFFTAG_FAXMODE, FAXMODE_NORTC);
        } else
                return (0);
}

/*
 * CCITT Group 3 1-D Modified Huffman RLE Compression Support.
 * (Compression algorithms 2 and 32771)
 */

/*
 * Decode the requested amount of RLE-encoded data.
 */
static int
```

---

kfax'Fax3DecodeRLE() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1396)

```
Fax3DecodeRLE(TIFF* tif, tidata_t buf, tsize_t occ, tsample_t s)
{
```

```
        DECLARE_STATE(tif, sp, "Fax3DecodeRLE");
        int mode = sp->b.mode;

        (void) s;
        CACHE_STATE(tif, sp);
        thisrun = sp->curruns;
        while ((long)occ > 0) {
                a0 = 0;
                RunLength = 0;
                pa = thisrun;
#ifdef FAX3_DEBUG
                printf("\nBitAcc=%08X, BitsAvail = %d\n", BitAcc, BitsAvail);
                printf("------------------- %d\n", tif->tif_row);
                fflush(stdout);
#endif
                EXPAND1D(EOFRLE);
                (*sp->fill)(buf, thisrun, pa, lastx);
                /*
                 * Cleanup at the end of the row.
                 */
                if (mode & FAXMODE_BYTEALIGN) {
                        int n = BitsAvail - (BitsAvail &~ 7);
                        ClrBits(n);
                } else if (mode & FAXMODE_WORDALIGN) {
                        int n = BitsAvail - (BitsAvail &~ 15);
                        ClrBits(n);
                        if (BitsAvail == 0 && !isAligned(cp, uint16))
                            cp++;
                }
                buf += sp->b.rowbytes;
                occ -= sp->b.rowbytes;
                if (occ != 0)
                        tif->tif_row++;
                continue;
        EOFRLE:                         /* premature EOF */
                (*sp->fill)(buf, thisrun, pa, lastx);
                UNCACHE_STATE(tif, sp);
                return (-1);
        }
        UNCACHE_STATE(tif, sp);
        return (1);
}

int
```

---

kfax'TIFFInitCCITTRLE() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1442)

```
TIFFInitCCITTRLE(TIFF* tif, int scheme)
{
        if (TIFFInitCCITTFax3(tif, scheme)) {   /* reuse G3 compression */
                tif->tif_decoderow = Fax3DecodeRLE;
                tif->tif_decodestrip = Fax3DecodeRLE;
                tif->tif_decodetile = Fax3DecodeRLE;
                /*
                 * Suppress RTC+EOLs when encoding and byte-align data.
                 */
                return TIFFSetField(tif, TIFFTAG_FAXMODE,
                    FAXMODE_NORTC|FAXMODE_NOEOL|FAXMODE_BYTEALIGN);
```

```
        } else
                return (0);
}

int
```

---

kfax'TIFFInitCCITTRLEW() (./kdegraphics/kfax/libtiffax/tif_fax3.c:1458)

```
TIFFInitCCITTRLEW(TIFF* tif, int scheme)
{
        if (TIFFInitCCITTFax3(tif, scheme)) {   /* reuse G3 compression */
                tif->tif_decoderow = Fax3DecodeRLE;
                tif->tif_decodestrip = Fax3DecodeRLE;
                tif->tif_decodetile = Fax3DecodeRLE;
                /*
                 * Suppress RTC+EOLs when encoding and word-align data.
                 */
                return TIFFSetField(tif, TIFFTAG_FAXMODE,
                    FAXMODE_NORTC|FAXMODE_NOEOL|FAXMODE_WORDALIGN);
        } else
                return (0);
}
```

---

kfax'TIFFFlush() (./kdegraphics/kfax/libtiffax/tif_flush.c:33)

```
TIFFFlush(TIFF* tif)
{

        if (tif->tif_mode != O_RDONLY) {
                if (!TIFFFlushData(tif))
                        return (0);
                if ((tif->tif_flags & TIFF_DIRTYDIRECT) &&
                    !TIFFWriteDirectory(tif))
                        return (0);
        }
        return (1);
}

/*
 * Flush buffered data to the file.
 */
int
```

---

kfax'TIFFFlushData() (./kdegraphics/kfax/libtiffax/tif_flush.c:50)

```
TIFFFlushData(TIFF* tif)
{
        if ((tif->tif_flags & TIFF_BEENWRITING) == 0)
                return (0);
        if (tif->tif_flags & TIFF_POSTENCODE) {
                tif->tif_flags &= ~TIFF_POSTENCODE;
                if (!(*tif->tif_postencode)(tif))
                        return (0);
```

```
        }
        return (TIFFFlushData1(tif));
}
```

---

kfax'TIFFRGBAImageOK() (./kdegraphics/kfax/libtiffax/tif_getimage.c:52)

```
TIFFRGBAImageOK(TIFF* tif, char emsg[1024])
{
    TIFFDirectory* td = &tif->tif_dir;
    uint16 photometric;
    int colorchannels;

    switch (td->td_bitspersample) {
    case 1: case 2: case 4:
    case 8: case 16:
        break;
    default:
        sprintf(emsg, "Sorry, can not handle images with %d-bit samples",
            td->td_bitspersample);
        return (0);
    }
    colorchannels = td->td_samplesperpixel - td->td_extrasamples;
    if (!TIFFGetField(tif, TIFFTAG_PHOTOMETRIC, &photometric)) {
        switch (colorchannels) {
        case 1:
            photometric = PHOTOMETRIC_MINISBLACK;
            break;
        case 3:
            photometric = PHOTOMETRIC_RGB;
            break;
        default:
            sprintf(emsg, "Missing needed %s tag", photoTag);
            return (0);
        }
    }
    switch (photometric) {
    case PHOTOMETRIC_MINISWHITE:
    case PHOTOMETRIC_MINISBLACK:
    case PHOTOMETRIC_PALETTE:
        if (td->td_planarconfig == PLANARCONFIG_CONTIG && td->td_samplesperpixel
            sprintf(emsg,
                "Sorry, can not handle contiguous data with %s=%d, and %s=%d",
                photoTag, photometric,
                "Samples/pixel", td->td_samplesperpixel);
            return (0);
        }
        break;
    case PHOTOMETRIC_YCBCR:
        if (td->td_planarconfig != PLANARCONFIG_CONTIG) {
            sprintf(emsg, "Sorry, can not handle YCbCr images with %s=%d",
                "Planarconfiguration", td->td_planarconfig);
            return (0);
        }
        break;
    case PHOTOMETRIC_RGB:
        if (colorchannels < 3) {
            sprintf(emsg, "Sorry, can not handle RGB image with %s=%d",
                "Color channels", colorchannels);
```

```
            return (0);
        }
        break;
#ifdef CMYK_SUPPORT
    case PHOTOMETRIC_SEPARATED:
        if (td->td_inkset != INKSET_CMYK) {
            sprintf(emsg, "Sorry, can not handle separated image with %s=%d",
                "InkSet", td->td_inkset);
            return (0);
        }
        if (td->td_samplesperpixel != 4) {
            sprintf(emsg, "Sorry, can not handle separated image with %s=%d",
                "Samples/pixel", td->td_samplesperpixel);
            return (0);
        }
        break;
#endif
    default:
        sprintf(emsg, "Sorry, can not handle image with %s=%d",
            photoTag, photometric);
        return (0);
    }
    return (1);
}

void
```

---

kfax'TIFFRGBAImageEnd() (./kdegraphics/kfax/libtiffax/tif_getimage.c:130)

```
TIFFRGBAImageEnd(TIFFRGBAImage* img)
{
    if (img->Map)
        _TIFFfree(img->Map), img->Map = NULL;
    if (img->BWmap)
        _TIFFfree(img->BWmap), img->BWmap = NULL;
    if (img->PALmap)
        _TIFFfree(img->PALmap), img->PALmap = NULL;
    if (img->ycbcr)
        _TIFFfree(img->ycbcr), img->ycbcr = NULL;
}

static int
```

---

kfax'isCCITTCompression() (./kdegraphics/kfax/libtiffax/tif_getimage.c:143)

```
isCCITTCompression(TIFF* tif)
{
    uint16 compress;
    TIFFGetField(tif, TIFFTAG_COMPRESSION, &compress);
    return (compress == COMPRESSION_CCITTFAX3 ||
            compress == COMPRESSION_CCITTFAX4 ||
            compress == COMPRESSION_CCITTRLE ||
            compress == COMPRESSION_CCITTRLEW);
}

int
```

kfax'TIFFRGBAImageBegin() (./kdegraphics/kfax/libtiffax/tif_getimage.c:154)

```
TIFFRGBAImageBegin(TIFFRGBAImage* img, TIFF* tif, int stop, char emsg[1024])
{
    uint16* sampleinfo;
    uint16 extrasamples;
    uint16 planarconfig;
    int colorchannels;

    img->tif = tif;
    img->stoponerr = stop;
    TIFFGetFieldDefaulted(tif, TIFFTAG_BITSPERSAMPLE, &img->bitspersample);
    switch (img->bitspersample) {
    case 1: case 2: case 4:
    case 8: case 16:
        break;
    default:
        sprintf(emsg, "Sorry, can not image with %d-bit samples",
            img->bitspersample);
        return (0);
    }
    img->alpha = 0;
    TIFFGetFieldDefaulted(tif, TIFFTAG_SAMPLESPERPIXEL, &img->samplesperpixel);
    TIFFGetFieldDefaulted(tif, TIFFTAG_EXTRASAMPLES,
        &extrasamples, &sampleinfo);
    if (extrasamples == 1)
        switch (sampleinfo[0]) {
        case EXTRASAMPLE_ASSOCALPHA:    /* data is pre-multiplied */
        case EXTRASAMPLE_UNASSALPHA:    /* data is not pre-multiplied */
            img->alpha = sampleinfo[0];
            break;
        }
    colorchannels = img->samplesperpixel - extrasamples;
    TIFFGetFieldDefaulted(tif, TIFFTAG_PLANARCONFIG, &planarconfig);
    if (!TIFFGetField(tif, TIFFTAG_PHOTOMETRIC, &img->photometric)) {
        switch (colorchannels) {
        case 1:
            if (isCCITTCompression(tif))
                img->photometric = PHOTOMETRIC_MINISWHITE;
            else
                img->photometric = PHOTOMETRIC_MINISBLACK;
            break;
        case 3:
            img->photometric = PHOTOMETRIC_RGB;
            break;
        default:
            sprintf(emsg, "Missing needed %s tag", photoTag);
            return (0);
        }
    }
    switch (img->photometric) {
    case PHOTOMETRIC_PALETTE:
        if (!TIFFGetField(tif, TIFFTAG_COLORMAP,
            &img->redcmap, &img->greencmap, &img->bluecmap)) {
            TIFFError(TIFFFileName(tif), "Missing required \"Colormap\" tag");
            return (0);
        }
```

```
        /* fall thru... */
    case PHOTOMETRIC_MINISWHITE:
    case PHOTOMETRIC_MINISBLACK:
        if (planarconfig == PLANARCONFIG_CONTIG && img->samplesperpixel != 1) {
            sprintf(emsg,
                "Sorry, can not handle contiguous data with %s=%d, and %s=%d",
                photoTag, img->photometric,
                "Samples/pixel", img->samplesperpixel);
            return (0);
        }
        break;
    case PHOTOMETRIC_YCBCR:
        if (planarconfig != PLANARCONFIG_CONTIG) {
            sprintf(emsg, "Sorry, can not handle YCbCr images with %s=%d",
                "Planarconfiguration", planarconfig);
            return (0);
        }
        /* It would probably be nice to have a reality check here. */
        { uint16 compress;
          TIFFGetField(tif, TIFFTAG_COMPRESSION, &compress);
          if (compress == COMPRESSION_JPEG && planarconfig == PLANARCONFIG_CONTI
            /* can rely on libjpeg to convert to RGB */
            /* XXX should restore current state on exit */
            TIFFSetField(tif, TIFFTAG_JPEGCOLORMODE, JPEGCOLORMODE_RGB);
            img->photometric = PHOTOMETRIC_RGB;
          }
        }
        break;
    case PHOTOMETRIC_RGB:
        if (colorchannels < 3) {
            sprintf(emsg, "Sorry, can not handle RGB image with %s=%d",
                "Color channels", colorchannels);
            return (0);
        }
        break;
    case PHOTOMETRIC_SEPARATED: {
        uint16 inkset;
        TIFFGetFieldDefaulted(tif, TIFFTAG_INKSET, &inkset);
        if (inkset != INKSET_CMYK) {
            sprintf(emsg, "Sorry, can not handle separated image with %s=%d",
                "InkSet", inkset);
            return (0);
        }
        if (img->samplesperpixel != 4) {
            sprintf(emsg, "Sorry, can not handle separated image with %s=%d",
                "Samples/pixel", img->samplesperpixel);
            return (0);
        }
        break;
    }
    default:
        sprintf(emsg, "Sorry, can not handle image with %s=%d",
            photoTag, img->photometric);
        return (0);
    }
    img->Map = NULL;
    img->BWmap = NULL;
    img->PALmap = NULL;
    img->ycbcr = NULL;
    TIFFGetField(tif, TIFFTAG_IMAGEWIDTH, &img->width);
    TIFFGetField(tif, TIFFTAG_IMAGELENGTH, &img->height);
```

```
    TIFFGetFieldDefaulted(tif, TIFFTAG_ORIENTATION, &img->orientation);
    img->isContig =
        !(planarconfig == PLANARCONFIG_SEPARATE && colorchannels > 1);
    if (img->isContig) {
        img->get = TIFFIsTiled(tif) ? gtTileContig : gtStripContig;
        (void) pickTileContigCase(img);
    } else {
        img->get = TIFFIsTiled(tif) ? gtTileSeparate : gtStripSeparate;
        (void) pickTileSeparateCase(img);
    }
    return (1);
}

int
```

---

kfax'TIFFRGBAImageGet() (./kdegraphics/kfax/libtiffax/tif_getimage.c:284)

```
TIFFRGBAImageGet(TIFFRGBAImage* img, uint32* raster, uint32 w, uint32 h)
{
    if (img->get == NULL) {
        TIFFError(TIFFFileName(img->tif), "No \"get\" routine setup");
        return (0);
    }
    if (img->put.any == NULL) {
        TIFFError(TIFFFileName(img->tif),
            "No \"put\" routine setupl; probably can not handle image format");
        return (0);
    }
    return (*img->get)(img, raster, w, h);
}

/*
 * Read the specified image into an ABGR-format raster.
 */
int
```

---

kfax'TIFFReadRGBAImage() (./kdegraphics/kfax/libtiffax/tif_getimage.c:302)

```
TIFFReadRGBAImage(TIFF* tif,
    uint32 rwidth, uint32 rheight, uint32* raster, int stop)
{
    char emsg[1024];
    TIFFRGBAImage img;
    int ok;

    if (TIFFRGBAImageBegin(&img, tif, stop, emsg)) {
        /* XXX verify rwidth and rheight against width and height */
        ok = TIFFRGBAImageGet(&img, raster+(rheight-img.height)*rwidth,
            rwidth, img.height);
        TIFFRGBAImageEnd(&img);
    } else {
        TIFFError(TIFFFileName(tif), emsg);
        ok = 0;
    }
    return (ok);
}
```

```
static uint32
```

---

kfax'setorientation() (./kdegraphics/kfax/libtiffax/tif_getimage.c:322)

```
setorientation(TIFFRGBAImage* img, uint32 h)
{
    TIFF* tif = img->tif;
    uint32 y;

    switch (img->orientation) {
    case ORIENTATION_BOTRIGHT:
    case ORIENTATION_RIGHTBOT:  /* XXX */
    case ORIENTATION_LEFTBOT:   /* XXX */
        TIFFWarning(TIFFFileName(tif), "using bottom-left orientation");
        img->orientation = ORIENTATION_BOTLEFT;
        /* fall thru... */
    case ORIENTATION_BOTLEFT:
        y = 0;
        break;
    case ORIENTATION_TOPRIGHT:
    case ORIENTATION_RIGHTTOP:  /* XXX */
    case ORIENTATION_LEFTTOP:   /* XXX */
    default:
        TIFFWarning(TIFFFileName(tif), "using top-left orientation");
        img->orientation = ORIENTATION_TOPLEFT;
        /* fall thru... */
    case ORIENTATION_TOPLEFT:
        y = h-1;
        break;
    }
    return (y);
}

/*
 * Get an tile-organized image that has
 *      PlanarConfiguration contiguous if SamplesPerPixel > 1
 * or
 *      SamplesPerPixel == 1
 */
static int
```

---

kfax'gtTileContig() (./kdegraphics/kfax/libtiffax/tif_getimage.c:358)

```
gtTileContig(TIFFRGBAImage* img, uint32* raster, uint32 w, uint32 h)
{
    TIFF* tif = img->tif;
    tileContigRoutine put = img->put.contig;
    uint16 orientation;
    uint32 col, row, y;
    uint32 tw, th;
    u_char* buf;
    int32 fromskew, toskew;
    uint32 nrow;

    buf = (u_char*) _TIFFmalloc(TIFFTileSize(tif));
```

```
    if (buf == 0) {
        TIFFError(TIFFFileName(tif), "No space for tile buffer");
        return (0);
    }
    TIFFGetField(tif, TIFFTAG_TILEWIDTH, &tw);
    TIFFGetField(tif, TIFFTAG_TILELENGTH, &th);
    y = setorientation(img, h);
    orientation = img->orientation;
    toskew = -(int32) (orientation == ORIENTATION_TOPLEFT ? tw+w : tw-w);
    for (row = 0; row < h; row += th) {
        nrow = (row + th > h ? h - row : th);
        for (col = 0; col < w; col += tw) {
            if (TIFFReadTile(tif, buf, col, row, 0, 0) < 0 && img->stoponerr)
                break;
            if (col + tw > w) {
                /*
                 * Tile is clipped horizontally.  Calculate
                 * visible portion and skewing factors.
                 */
                uint32 npix = w - col;
                fromskew = tw - npix;
                (*put)(img, raster+y*w+col, col, y,
                    npix, nrow, fromskew, toskew + fromskew, buf);
            } else {
                (*put)(img, raster+y*w+col, col, y, tw, nrow, 0, toskew, buf);
            }
        }
        y += (orientation == ORIENTATION_TOPLEFT ?
            -(int32) nrow : (int32) nrow);
    }
    _TIFFfree(buf);
    return (1);
}

/*
 * Get an tile-organized image that has
 *       SamplesPerPixel > 1
 *       PlanarConfiguration separated
 * We assume that all such images are RGB.
 */
static int
```

---

kfax'gtTileSeparate() (./kdegraphics/kfax/libtiffax/tif_getimage.c:411)

```
gtTileSeparate(TIFFRGBAImage* img, uint32* raster, uint32 w, uint32 h)
{
    TIFF* tif = img->tif;
    tileSeparateRoutine put = img->put.separate;
    uint16 orientation;
    uint32 col, row, y;
    uint32 tw, th;
    u_char* buf;
    u_char* r;
    u_char* g;
    u_char* b;
    u_char* a;
    tsize_t tilesize;
    int32 fromskew, toskew;
```

```
    int alpha = img->alpha;
    uint32 nrow;

    tilesize = TIFFTileSize(tif);
    buf = (u_char*) _TIFFmalloc(4*tilesize);
    if (buf == 0) {
        TIFFError(TIFFFileName(tif), "No space for tile buffer");
        return (0);
    }
    r = buf;
    g = r + tilesize;
    b = g + tilesize;
    a = b + tilesize;
    if (!alpha)
        memset(a, 0xff, tilesize);
    TIFFGetField(tif, TIFFTAG_TILEWIDTH, &tw);
    TIFFGetField(tif, TIFFTAG_TILELENGTH, &th);
    y = setorientation(img, h);
    orientation = img->orientation;
    toskew = -(int32) (orientation == ORIENTATION_TOPLEFT ? tw+w : tw-w);
    for (row = 0; row < h; row += th) {
        nrow = (row + th > h ? h - row : th);
        for (col = 0; col < w; col += tw) {
            if (TIFFReadTile(tif, r, col, row,0,0) < 0 && img->stoponerr)
                break;
            if (TIFFReadTile(tif, g, col, row,0,1) < 0 && img->stoponerr)
                break;
            if (TIFFReadTile(tif, b, col, row,0,2) < 0 && img->stoponerr)
                break;
            if (alpha && TIFFReadTile(tif,a,col,row,0,3) < 0 && img->stoponerr)
                break;
            if (col + tw > w) {
                /*
                 * Tile is clipped horizontally.  Calculate
                 * visible portion and skewing factors.
                 */
                uint32 npix = w - col;
                fromskew = tw - npix;
                (*put)(img, raster+y*w+col, col, y,
                    npix, nrow, fromskew, toskew + fromskew, r, g, b, a);
            } else {
                (*put)(img, raster+y*w+col, col, y,
                    tw, nrow, 0, toskew, r, g, b, a);
            }
        }
        y += (orientation == ORIENTATION_TOPLEFT ?
            -(int32) nrow : (int32) nrow);
    }
    _TIFFfree(buf);
    return (1);
}

/*
 * Get a strip-organized image that has
 *      PlanarConfiguration contiguous if SamplesPerPixel > 1
 * or
 *      SamplesPerPixel == 1
 */
static int
```

kfax'gtStripContig() (./kdegraphics/kfax/libtiffax/tif_getimage.c:484)

```
gtStripContig(TIFFRGBAImage* img, uint32* raster, uint32 w, uint32 h)
{
    TIFF* tif = img->tif;
    tileContigRoutine put = img->put.contig;
    uint16 orientation;
    uint32 row, y, nrow;
    u_char* buf;
    uint32 rowsperstrip;
    uint32 imagewidth = img->width;
    tsize_t scanline;
    int32 fromskew, toskew;

    buf = (u_char*) _TIFFmalloc(TIFFStripSize(tif));
    if (buf == 0) {
        TIFFError(TIFFFileName(tif), "No space for strip buffer");
        return (0);
    }
    y = setorientation(img, h);
    orientation = img->orientation;
    toskew = -(int32) (orientation == ORIENTATION_TOPLEFT ? w+w : w-w);
    TIFFGetFieldDefaulted(tif, TIFFTAG_ROWSPERSTRIP, &rowsperstrip);
    scanline = TIFFScanlineSize(tif);
    fromskew = (w < imagewidth ? imagewidth - w : 0);
    for (row = 0; row < h; row += rowsperstrip) {
        nrow = (row + rowsperstrip > h ? h - row : rowsperstrip);
        if (TIFFReadEncodedStrip(tif, TIFFComputeStrip(tif, row, 0),
            buf, nrow*scanline) < 0 && img->stoponerr)
                break;
        (*put)(img, raster+y*w, 0, y, w, nrow, fromskew, toskew, buf);
        y += (orientation == ORIENTATION_TOPLEFT ?
            -(int32) nrow : (int32) nrow);
    }
    _TIFFfree(buf);
    return (1);
}

/*
 * Get a strip-organized image with
 *      SamplesPerPixel > 1
 *      PlanarConfiguration separated
 * We assume that all such images are RGB.
 */
static int
```

---

kfax'gtStripSeparate() (./kdegraphics/kfax/libtiffax/tif_getimage.c:527)

```
gtStripSeparate(TIFFRGBAImage* img, uint32* raster, uint32 w, uint32 h)
{
    TIFF* tif = img->tif;
    tileSeparateRoutine put = img->put.separate;
    uint16 orientation;
    u_char *buf;
    u_char *r, *g, *b, *a;
    uint32 row, y, nrow;
    tsize_t scanline;
```

```
    uint32 rowsperstrip;
    uint32 imagewidth = img->width;
    tsize_t stripsize;
    int32 fromskew, toskew;
    int alpha = img->alpha;

    stripsize = TIFFStripSize(tif);
    r = buf = (u_char *)_TIFFmalloc(4*stripsize);
    if (buf == 0) {
        TIFFError(TIFFFileName(tif), "No space for tile buffer");
        return (0);
    }
    g = r + stripsize;
    b = g + stripsize;
    a = b + stripsize;
    if (!alpha)
        memset(a, 0xff, stripsize);
    y = setorientation(img, h);
    orientation = img->orientation;
    toskew = -(int32) (orientation == ORIENTATION_TOPLEFT ? w+w : w-w);
    TIFFGetFieldDefaulted(tif, TIFFTAG_ROWSPERSTRIP, &rowsperstrip);
    scanline = TIFFScanlineSize(tif);
    fromskew = (w < imagewidth ? imagewidth - w : 0);
    for (row = 0; row < h; row += rowsperstrip) {
        nrow = (row + rowsperstrip > h ? h - row : rowsperstrip);
        if (TIFFReadEncodedStrip(tif, TIFFComputeStrip(tif, row, 0),
            r, nrow*scanline) < 0 && img->stoponerr)
            break;
        if (TIFFReadEncodedStrip(tif, TIFFComputeStrip(tif, row, 1),
            g, nrow*scanline) < 0 && img->stoponerr)
            break;
        if (TIFFReadEncodedStrip(tif, TIFFComputeStrip(tif, row, 2),
            b, nrow*scanline) < 0 && img->stoponerr)
            break;
        if (alpha &&
            (TIFFReadEncodedStrip(tif, TIFFComputeStrip(tif, row, 3),
            a, nrow*scanline) < 0 && img->stoponerr))
            break;
        (*put)(img, raster+y*w, 0, y, w, nrow, fromskew, toskew, r, g, b, a);
        y += (orientation == ORIENTATION_TOPLEFT ?
            -(int32) nrow : (int32) nrow);
    }
    _TIFFfree(buf);
    return (1);
}

/*
 * The following routines move decoded data returned
 * from the TIFF library into rasters filled with packed
 * ABGR pixels (i.e. suitable for passing to lrecwrite.)
 *
 * The routines have been created according to the most
 * important cases and optimized.  pickTileContigCase and
 * pickTileSeparateCase analyze the parameters and select
 * the appropriate "put" routine to use.
 */
```

---

kfax'DECLAREContigPutFunc()

(./kdegraphics/kfax/libtiffax/tif_getimage.c:665)

```
DECLAREContigPutFunc(put8bitcmaptile)
{
    uint32** PALmap = img->PALmap;

    (void) x; (void) y;
    while (h-- > 0) {
        UNROLL8(w, NOP, *cp++ = PALmap[*pp++][0]);
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 4-bit palette => colormap/RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:680)

```
DECLAREContigPutFunc(put4bitcmaptile)
{
    uint32** PALmap = img->PALmap;

    (void) x; (void) y;
    fromskew /= 2;
    while (h-- > 0) {
        uint32* bw;
        UNROLL2(w, bw = PALmap[*pp++], *cp++ = *bw++);
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 2-bit palette => colormap/RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:697)

```
DECLAREContigPutFunc(put2bitcmaptile)
{
    uint32** PALmap = img->PALmap;

    (void) x; (void) y;
    fromskew /= 4;
    while (h-- > 0) {
        uint32* bw;
        UNROLL4(w, bw = PALmap[*pp++], *cp++ = *bw++);
        cp += toskew;
        pp += fromskew;
    }
}
```

```
/*
 * 1-bit palette => colormap/RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:714)

```
DECLAREContigPutFunc(put1bitcmaptile)
{
    uint32** PALmap = img->PALmap;

    (void) x; (void) y;
    fromskew /= 8;
    while (h-- > 0) {
        uint32* bw;
        UNROLL8(w, bw = PALmap[*pp++], *cp++ = *bw++);
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 8-bit greyscale => colormap/RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:731)

```
DECLAREContigPutFunc(putgreytile)
{
    uint32** BWmap = img->BWmap;

    (void) y;
    while (h-- > 0) {
        for (x = w; x-- > 0;)
            *cp++ = BWmap[*pp++][0];
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 1-bit bilevel => colormap/RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:747)

```
DECLAREContigPutFunc(put1bitbwtile)
{
    uint32** BWmap = img->BWmap;
```

```
    (void) x; (void) y;
    fromskew /= 8;
    while (h-- > 0) {
        uint32* bw;
        UNROLL8(w, bw = BWmap[*pp++], *cp++ = *bw++);
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 2-bit greyscale => colormap/RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:764)

```
DECLAREContigPutFunc(put2bitbwtile)
{
    uint32** BWmap = img->BWmap;

    (void) x; (void) y;
    fromskew /= 4;
    while (h-- > 0) {
        uint32* bw;
        UNROLL4(w, bw = BWmap[*pp++], *cp++ = *bw++);
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 4-bit greyscale => colormap/RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:781)

```
DECLAREContigPutFunc(put4bitbwtile)
{
    uint32** BWmap = img->BWmap;

    (void) x; (void) y;
    fromskew /= 2;
    while (h-- > 0) {
        uint32* bw;
        UNROLL2(w, bw = BWmap[*pp++], *cp++ = *bw++);
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 8-bit packed samples, no Map => RGB
```

```
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:798)

```
DECLAREContigPutFunc(putRGBcontig8bittile)
{
    int samplesperpixel = img->samplesperpixel;

    (void) x; (void) y;
    fromskew *= samplesperpixel;
    while (h-- > 0) {
        UNROLL8(w, NOP,
            *cp++ = PACK(pp[0], pp[1], pp[2]);
            pp += samplesperpixel);
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 8-bit packed samples, w/ Map => RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:816)

```
DECLAREContigPutFunc(putRGBcontig8bitMaptile)
{
    TIFFRGBValue* Map = img->Map;
    int samplesperpixel = img->samplesperpixel;

    (void) y;
    fromskew *= samplesperpixel;
    while (h-- > 0) {
        for (x = w; x-- > 0;) {
            *cp++ = PACK(Map[pp[0]], Map[pp[1]], Map[pp[2]]);
            pp += samplesperpixel;
        }
        pp += fromskew;
        cp += toskew;
    }
}

/*
 * 8-bit packed samples => RGBA w/ associated alpha
 * (known to have Map == NULL)
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:837)

```
DECLAREContigPutFunc(putRGBAAcontig8bittile)
{
    int samplesperpixel = img->samplesperpixel;

    (void) x; (void) y;
    fromskew *= samplesperpixel;
    while (h-- > 0) {
        UNROLL8(w, NOP,
            *cp++ = PACK4(pp[0], pp[1], pp[2], pp[3]);
            pp += samplesperpixel);
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 8-bit packed samples => RGBA w/ unassociated alpha
 * (known to have Map == NULL)
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:856)

```
DECLAREContigPutFunc(putRGBUAcontig8bittile)
{
    int samplesperpixel = img->samplesperpixel;

    (void) y;
    fromskew *= samplesperpixel;
    while (h-- > 0) {
        uint32 r, g, b, a;
        for (x = w; x-- > 0;) {
            a = pp[3];
            r = (pp[0] * a) / 255;
            g = (pp[1] * a) / 255;
            b = (pp[2] * a) / 255;
            *cp++ = PACK4(r,g,b,a);
            pp += samplesperpixel;
        }
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 16-bit packed samples => RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:880)

```
DECLAREContigPutFunc(putRGBcontig16bittile)
{
    int samplesperpixel = img->samplesperpixel;
```

```
    uint16 *wp = (uint16 *)pp;

    (void) y;
    fromskew *= samplesperpixel;
    while (h-- > 0) {
        for (x = w; x-- > 0;) {
            *cp++ = PACKW(wp[0], wp[1], wp[2]);
            wp += samplesperpixel;
        }
        cp += toskew;
        wp += fromskew;
    }
}

/*
 * 16-bit packed samples => RGBA w/ associated alpha
 * (known to have Map == NULL)
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:901)

```
DECLAREContigPutFunc(putRGBAAcontig16bittile)
{
    int samplesperpixel = img->samplesperpixel;
    uint16 *wp = (uint16 *)pp;

    (void) y;
    fromskew *= samplesperpixel;
    while (h-- > 0) {
        for (x = w; x-- > 0;) {
            *cp++ = PACKW4(wp[0], wp[1], wp[2], wp[3]);
            wp += samplesperpixel;
        }
        cp += toskew;
        wp += fromskew;
    }
}

/*
 * 16-bit packed samples => RGBA w/ unassociated alpha
 * (known to have Map == NULL)
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:922)

```
DECLAREContigPutFunc(putRGBUAcontig16bittile)
{
    int samplesperpixel = img->samplesperpixel;
    uint16 *wp = (uint16 *)pp;

    (void) y;
    fromskew *= samplesperpixel;
    while (h-- > 0) {
```

```
        uint32 r,g,b,a;
        /*
         * We shift alpha down four bits just in case unsigned
         * arithmetic doesn't handle the full range.
         * We still have plenty of accuracy, since the output is 8 bits.
         * So we have (r * 0xffff) * (a * 0xfff)) = r*a * (0xffff*0xfff)
         * Since we want r*a * 0xff for eight bit output,
         * we divide by (0xffff * 0xfff) / 0xff == 0x10eff.
         */
        for (x = w; x-- > 0;) {
            a = wp[3] >> 4;
            r = (wp[0] * a) / 0x10eff;
            g = (wp[1] * a) / 0x10eff;
            b = (wp[2] * a) / 0x10eff;
            *cp++ = PACK4(r,g,b,a);
            wp += samplesperpixel;
        }
        cp += toskew;
        wp += fromskew;
    }
}

/*
 * 8-bit packed CMYK samples w/o Map => RGB
 *
 * NB: The conversion of CMYK->RGB is *very* crude.
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:957)

```
DECLAREContigPutFunc(putRGBcontig8bitCMYKtile)
{
    int samplesperpixel = img->samplesperpixel;
    uint16 r, g, b, k;

    (void) x; (void) y;
    fromskew *= samplesperpixel;
    while (h-- > 0) {
        UNROLL8(w, NOP,
            k = 255 - pp[3];
            r = (k*(255-pp[0]))/255;
            g = (k*(255-pp[1]))/255;
            b = (k*(255-pp[2]))/255;
            *cp++ = PACK(r, g, b);
            pp += samplesperpixel);
        cp += toskew;
        pp += fromskew;
    }
}

/*
 * 8-bit packed CMYK samples w/Map => RGB
 *
 * NB: The conversion of CMYK->RGB is *very* crude.
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:982)

```
DECLAREContigPutFunc(putRGBcontig8bitCMYKMaptile)
{
    int samplesperpixel = img->samplesperpixel;
    TIFFRGBValue* Map = img->Map;
    uint16 r, g, b, k;

    (void) y;
    fromskew *= samplesperpixel;
    while (h-- > 0) {
        for (x = w; x-- > 0;) {
            k = 255 - pp[3];
            r = (k*(255-pp[0]))/255;
            g = (k*(255-pp[1]))/255;
            b = (k*(255-pp[2]))/255;
            *cp++ = PACK(Map[r], Map[g], Map[b]);
            pp += samplesperpixel;
        }
        pp += fromskew;
        cp += toskew;
    }
}
```

---

kfax'DECLARESepPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1017)

```
DECLARESepPutFunc(putRGBseparate8bittile)
{
    (void) img; (void) x; (void) y; (void) a;
    while (h-- > 0) {
        UNROLL8(w, NOP, *cp++ = PACK(*r++, *g++, *b++));
        SKEW(r, g, b, fromskew);
        cp += toskew;
    }
}

/*
 * 8-bit unpacked samples => RGB
 */
```

---

kfax'DECLARESepPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1030)

```
DECLARESepPutFunc(putRGBseparate8bitMaptile)
{
    TIFFRGBValue* Map = img->Map;

    (void) y; (void) a;
    while (h-- > 0) {
        for (x = w; x > 0; x--)
            *cp++ = PACK(Map[*r++], Map[*g++], Map[*b++]);
        SKEW(r, g, b, fromskew);
        cp += toskew;
```

```
    }
}

/*
 * 8-bit unpacked samples => RGBA w/ associated alpha
 */
```

---

kfax'DECLARESepPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1046)

```
DECLARESepPutFunc(putRGBAAseparate8bittile)
{
    (void) img; (void) x; (void) y;
    while (h-- > 0) {
        UNROLL8(w, NOP, *cp++ = PACK4(*r++, *g++, *b++, *a++));
        SKEW4(r, g, b, a, fromskew);
        cp += toskew;
    }
}

/*
 * 8-bit unpacked samples => RGBA w/ unassociated alpha
 */
```

---

kfax'DECLARESepPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1059)

```
DECLARESepPutFunc(putRGBUAseparate8bittile)
{
    (void) img; (void) y;
    while (h-- > 0) {
        uint32 rv, gv, bv, av;
        for (x = w; x-- > 0;) {
            av = *a++;
            rv = (*r++ * av) / 255;
            gv = (*g++ * av) / 255;
            bv = (*b++ * av) / 255;
            *cp++ = PACK4(rv,gv,bv,av);
        }
        SKEW4(r, g, b, a, fromskew);
        cp += toskew;
    }
}

/*
 * 16-bit unpacked samples => RGB
 */
```

---

kfax'DECLARESepPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1079)

```
DECLARESepPutFunc(putRGBseparate16bittile)
{
    uint16 *wr = (uint16*) r;
    uint16 *wg = (uint16*) g;
    uint16 *wb = (uint16*) b;
```

```
    (void) img; (void) y; (void) a;
    while (h-- > 0) {
        for (x = 0; x < w; x++)
            *cp++ = PACKW(*wr++, *wg++, *wb++);
        SKEW(wr, wg, wb, fromskew);
        cp += toskew;
    }
}

/*
 * 16-bit unpacked samples => RGBA w/ associated alpha
 */
```

---

kfax'DECLARESepPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1097)

```
DECLARESepPutFunc(putRGBAAseparate16bittile)
{
    uint16 *wr = (uint16*) r;
    uint16 *wg = (uint16*) g;
    uint16 *wb = (uint16*) b;
    uint16 *wa = (uint16*) a;

    (void) img; (void) y;
    while (h-- > 0) {
        for (x = 0; x < w; x++)
            *cp++ = PACKW4(*wr++, *wg++, *wb++, *wa++);
        SKEW4(wr, wg, wb, wa, fromskew);
        cp += toskew;
    }
}

/*
 * 16-bit unpacked samples => RGBA w/ unassociated alpha
 */
```

---

kfax'DECLARESepPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1116)

```
DECLARESepPutFunc(putRGBUAseparate16bittile)
{
    uint16 *wr = (uint16*) r;
    uint16 *wg = (uint16*) g;
    uint16 *wb = (uint16*) b;
    uint16 *wa = (uint16*) a;

    (void) img; (void) y;
    while (h-- > 0) {
        uint32 r,g,b,a;
        /*
         * We shift alpha down four bits just in case unsigned
         * arithmetic doesn't handle the full range.
         * We still have plenty of accuracy, since the output is 8 bits.
         * So we have (r * 0xffff) * (a * 0xfff)) = r*a * (0xffff*0xfff)
         * Since we want r*a * 0xff for eight bit output,
         * we divide by (0xffff * 0xfff) / 0xff == 0x10eff.
         */
```

```
        for (x = w; x-- > 0;) {
            a = *wa++ >> 4;
            r = (*wr++ * a) / 0x10eff;
            g = (*wg++ * a) / 0x10eff;
            b = (*wb++ * a) / 0x10eff;
            *cp++ = PACK4(r,g,b,a);
        }
        SKEW4(wr, wg, wb, wa, fromskew);
        cp += toskew;
    }
}

/*
 * YCbCr -> RGB conversion and packing routines.  The colorspace
 * conversion algorithm comes from the IJG v5a code; see below
 * for more information on how it works.
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1170)

```
DECLAREContigPutFunc(putcontig8bitYCbCr44tile)
{
    YCbCrSetup;
    uint32* cp1 = cp+w+toskew;
    uint32* cp2 = cp1+w+toskew;
    uint32* cp3 = cp2+w+toskew;
    u_int incr = 3*w+4*toskew;

    (void) y;
    /* XXX adjust fromskew */
    for (; h >= 4; h -= 4) {
        x = w>>2;
        do {
            int Cb = pp[16];
            int Cr = pp[17];

            YCbCrtoRGB(cp [0], pp[ 0]);
            YCbCrtoRGB(cp [1], pp[ 1]);
            YCbCrtoRGB(cp [2], pp[ 2]);
            YCbCrtoRGB(cp [3], pp[ 3]);
            YCbCrtoRGB(cp1[0], pp[ 4]);
            YCbCrtoRGB(cp1[1], pp[ 5]);
            YCbCrtoRGB(cp1[2], pp[ 6]);
            YCbCrtoRGB(cp1[3], pp[ 7]);
            YCbCrtoRGB(cp2[0], pp[ 8]);
            YCbCrtoRGB(cp2[1], pp[ 9]);
            YCbCrtoRGB(cp2[2], pp[10]);
            YCbCrtoRGB(cp2[3], pp[11]);
            YCbCrtoRGB(cp3[0], pp[12]);
            YCbCrtoRGB(cp3[1], pp[13]);
            YCbCrtoRGB(cp3[2], pp[14]);
            YCbCrtoRGB(cp3[3], pp[15]);

            cp += 4, cp1 += 4, cp2 += 4, cp3 += 4;
            pp += 18;
        } while (--x);
```

```
        cp += incr, cp1 += incr, cp2 += incr, cp3 += incr;
        pp += fromskew;
    }
}

/*
 * 8-bit packed YCbCr samples w/ 4,2 subsampling => RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1214)

```
DECLAREContigPutFunc(putcontig8bitYCbCr42tile)
{
    YCbCrSetup;
    uint32* cp1 = cp+w+toskew;
    u_int incr = 2*toskew+w;

    (void) y;
    /* XXX adjust fromskew */
    for (; h >= 2; h -= 2) {
        x = w>>2;
        do {
            int Cb = pp[8];
            int Cr = pp[9];

            YCbCrtoRGB(cp [0], pp[0]);
            YCbCrtoRGB(cp [1], pp[1]);
            YCbCrtoRGB(cp [2], pp[2]);
            YCbCrtoRGB(cp [3], pp[3]);
            YCbCrtoRGB(cp1[0], pp[4]);
            YCbCrtoRGB(cp1[1], pp[5]);
            YCbCrtoRGB(cp1[2], pp[6]);
            YCbCrtoRGB(cp1[3], pp[7]);

            cp += 4, cp1 += 4;
            pp += 10;
        } while (--x);
        cp += incr, cp1 += incr;
        pp += fromskew;
    }
}

/*
 * 8-bit packed YCbCr samples w/ 4,1 subsampling => RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1248)

```
DECLAREContigPutFunc(putcontig8bitYCbCr41tile)
{
    YCbCrSetup;

    (void) y;
```

```
    /* XXX adjust fromskew */
    do {
        x = w>>2;
        do {
            int Cb = pp[4];
            int Cr = pp[5];

            YCbCrtoRGB(cp [0], pp[0]);
            YCbCrtoRGB(cp [1], pp[1]);
            YCbCrtoRGB(cp [2], pp[2]);
            YCbCrtoRGB(cp [3], pp[3]);

            cp += 4;
            pp += 6;
        } while (--x);
        cp += toskew;
        pp += fromskew;
    } while (--h);
}

/*
 * 8-bit packed YCbCr samples w/ 2,2 subsampling => RGB
 */
```

---

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1276)

```
DECLAREContigPutFunc(putcontig8bitYCbCr22tile)
{
    YCbCrSetup;
    uint32* cp1 = cp+w+toskew;
    u_int incr = 2*toskew+w;

    (void) y;
    /* XXX adjust fromskew */
    for (; h >= 2; h -= 2) {
        x = w>>1;
        do {
            int Cb = pp[4];
            int Cr = pp[5];

            YCbCrtoRGB(cp [0], pp[0]);
            YCbCrtoRGB(cp [1], pp[1]);
            YCbCrtoRGB(cp1[0], pp[2]);
            YCbCrtoRGB(cp1[1], pp[3]);

            cp += 2, cp1 += 2;
            pp += 6;
        } while (--x);
        cp += incr, cp1 += incr;
        pp += fromskew;
    }
}

/*
 * 8-bit packed YCbCr samples w/ 2,1 subsampling => RGB
 */
```

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1306)

```
DECLAREContigPutFunc(putcontig8bitYCbCr21tile)
{
    YCbCrSetup;

    (void) y;
    /* XXX adjust fromskew */
    do {
        x = w>>1;
        do {
            int Cb = pp[2];
            int Cr = pp[3];

            YCbCrtoRGB(cp[0], pp[0]);
            YCbCrtoRGB(cp[1], pp[1]);

            cp += 2;
            pp += 4;
        } while (--x);
        cp += toskew;
        pp += fromskew;
    } while (--h);
}

/*
 * 8-bit packed YCbCr samples w/ no subsampling => RGB
 */
```

kfax'DECLAREContigPutFunc() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1332)

```
DECLAREContigPutFunc(putcontig8bitYCbCr11tile)
{
    YCbCrSetup;

    (void) y;
    /* XXX adjust fromskew */
    do {
        x = w>>1;
        do {
            int Cb = pp[1];
            int Cr = pp[2];

            YCbCrtoRGB(*cp++, pp[0]);

            pp += 3;
        } while (--x);
        cp += toskew;
        pp += fromskew;
    } while (--h);
}
```

kfax'TIFFYCbCrToRGBInit() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1379)

```
TIFFYCbCrToRGBInit(TIFFYCbCrToRGB* ycbcr, TIFF* tif)
{
    TIFFRGBValue* clamptab;
    float* coeffs;
    int i;

    clamptab = (TIFFRGBValue*)(
        (tidata_t) ycbcr+TIFFroundup(sizeof (TIFFYCbCrToRGB), sizeof (long)));
    _TIFFmemset(clamptab, 0, 256);              /* v < 0 => 0 */
    ycbcr->clamptab = (clamptab += 256);
    for (i = 0; i < 256; i++)
        clamptab[i] = i;
    _TIFFmemset(clamptab+256, 255, 2*256);      /* v > 255 => 255 */
    TIFFGetFieldDefaulted(tif, TIFFTAG_YCBCRCOEFFICIENTS, &coeffs);
    _TIFFmemcpy(ycbcr->coeffs, coeffs, 3*sizeof (float));
    { float f1 = 2-2*LumaRed;           int32 D1 = FIX(f1);
      float f2 = LumaRed*f1/LumaGreen;  int32 D2 = -FIX(f2);
      float f3 = 2-2*LumaBlue;          int32 D3 = FIX(f3);
      float f4 = LumaBlue*f3/LumaGreen; int32 D4 = -FIX(f4);
      int x;

      ycbcr->Cr_r_tab = (int*) (clamptab + 3*256);
      ycbcr->Cb_b_tab = ycbcr->Cr_r_tab + 256;
      ycbcr->Cr_g_tab = (int32*) (ycbcr->Cb_b_tab + 256);
      ycbcr->Cb_g_tab = ycbcr->Cr_g_tab + 256;
      /*
       * i is the actual input pixel value in the range 0..255
       * Cb and Cr values are in the range -128..127 (actually
       * they are in a range defined by the ReferenceBlackWhite
       * tag) so there is some range shifting to do here when
       * constructing tables indexed by the raw pixel data.
       *
       * XXX handle ReferenceBlackWhite correctly to calculate
       *     Cb/Cr values to use in constructing the tables.
       */
      for (i = 0, x = -128; i < 256; i++, x++) {
          ycbcr->Cr_r_tab[i] = (int)((D1*x + ONE_HALF)>>SHIFT);
          ycbcr->Cb_b_tab[i] = (int)((D3*x + ONE_HALF)>>SHIFT);
          ycbcr->Cr_g_tab[i] = D2*x;
          ycbcr->Cb_g_tab[i] = D4*x + ONE_HALF;
      }
    }
}
```

kfax'initYCbCrConversion() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1430)

```
initYCbCrConversion(TIFFRGBAImage* img)
{
    uint16 hs, vs;

    if (img->ycbcr == NULL) {
        img->ycbcr = (TIFFYCbCrToRGB*) _TIFFmalloc(
              TIFFroundup(sizeof (TIFFYCbCrToRGB), sizeof (long))
```

```
            + 4*256*sizeof (TIFFRGBValue)
            + 2*256*sizeof (int)
            + 2*256*sizeof (int32)
        );
        if (img->ycbcr == NULL) {
            TIFFError(TIFFFileName(img->tif),
                "No space for YCbCr->RGB conversion state");
            return (NULL);
        }
        TIFFYCbCrToRGBInit(img->ycbcr, img->tif);
    } else {
        float* coeffs;

        TIFFGetFieldDefaulted(img->tif, TIFFTAG_YCBCRCOEFFICIENTS, &coeffs);
        if (_TIFFmemcmp(coeffs, img->ycbcr->coeffs, 3*sizeof (float)) != 0)
            TIFFYCbCrToRGBInit(img->ycbcr, img->tif);
    }
    /*
     * The 6.0 spec says that subsampling must be
     * one of 1, 2, or 4, and that vertical subsampling
     * must always be <= horizontal subsampling; so
     * there are only a few possibilities and we just
     * enumerate the cases.
     */
    TIFFGetFieldDefaulted(img->tif, TIFFTAG_YCBCRSUBSAMPLING, &hs, &vs);
    switch ((hs<<4)|vs) {
    case 0x44: return (putcontig8bitYCbCr44tile);
    case 0x42: return (putcontig8bitYCbCr42tile);
    case 0x41: return (putcontig8bitYCbCr41tile);
    case 0x22: return (putcontig8bitYCbCr22tile);
    case 0x21: return (putcontig8bitYCbCr21tile);
    case 0x11: return (putcontig8bitYCbCr11tile);
    }
    return (NULL);
}

/*
 * Greyscale images with less than 8 bits/sample are handled
 * with a table to avoid lots of shifts and masks.  The table
 * is setup so that put*bwtile (below) can retrieve 8/bitspersample
 * pixel values simply by indexing into the table with one
 * number.
 */
static int
```

---

kfax'makebwmap() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1481)

```
makebwmap(TIFFRGBAImage* img)
{
    TIFFRGBValue* Map = img->Map;
    int bitspersample = img->bitspersample;
    int nsamples = 8 / bitspersample;
    int i;
    uint32* p;

    img->BWmap = (uint32**) _TIFFmalloc(
        256*sizeof (uint32 *)+(256*nsamples*sizeof(uint32)));
    if (img->BWmap == NULL) {
```

```
        TIFFError(TIFFFileName(img->tif), "No space for B&W mapping table");
        return (0);
    }
    p = (uint32*)(img->BWmap + 256);
    for (i = 0; i < 256; i++) {
        TIFFRGBValue c;
        img->BWmap[i] = p;
        switch (bitspersample) {
```

---

kfax'setupMap() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1536)

```
setupMap(TIFFRGBAImage* img)
{
    int32 x, range;

    range = (int32)((1L<<img->bitspersample)-1);
    img->Map = (TIFFRGBValue*) _TIFFmalloc((range+1) * sizeof (TIFFRGBValue));
    if (img->Map == NULL) {
        TIFFError(TIFFFileName(img->tif),
            "No space for photometric conversion table");
        return (0);
    }
    if (img->photometric == PHOTOMETRIC_MINISWHITE) {
        for (x = 0; x <= range; x++)
            img->Map[x] = ((range - x) * 255) / range;
    } else {
        for (x = 0; x <= range; x++)
            img->Map[x] = (x * 255) / range;
    }
    if (img->bitspersample <= 8 &&
        (img->photometric == PHOTOMETRIC_MINISBLACK ||
         img->photometric == PHOTOMETRIC_MINISWHITE)) {
        /*
         * Use photometric mapping table to construct
         * unpacking tables for samples <= 8 bits.
         */
        if (!makebwmap(img))
            return (0);
        /* no longer need Map, free it */
        _TIFFfree(img->Map), img->Map = NULL;
    }
    return (1);
}

static int
```

---

kfax'checkcmap() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1570)

```
checkcmap(TIFFRGBAImage* img)
{
    uint16* r = img->redcmap;
    uint16* g = img->greencmap;
    uint16* b = img->bluecmap;
    long n = 1L<<img->bitspersample;

    while (n-- > 0)
```

```
        if (*r++ >= 256 || *g++ >= 256 || *b++ >= 256)
            return (16);
    return (8);
}

static void
```

---

kfax'cvtcmap() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1584)

```
cvtcmap(TIFFRGBAImage* img)
{
    uint16* r = img->redcmap;
    uint16* g = img->greencmap;
    uint16* b = img->bluecmap;
    long i;

    for (i = (1L<<img->bitspersample)-1; i >= 0; i--) {
```

---

kfax'makecmap() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1608)

```
makecmap(TIFFRGBAImage* img)
{
    int bitspersample = img->bitspersample;
    int nsamples = 8 / bitspersample;
    uint16* r = img->redcmap;
    uint16* g = img->greencmap;
    uint16* b = img->bluecmap;
    uint32 *p;
    int i;

    img->PALmap = (uint32**) _TIFFmalloc(
        256*sizeof (uint32 *)+(256*nsamples*sizeof(uint32)));
    if (img->PALmap == NULL) {
        TIFFError(TIFFFileName(img->tif), "No space for Palette mapping table");
        return (0);
    }
    p = (uint32*)(img->PALmap + 256);
    for (i = 0; i < 256; i++) {
        TIFFRGBValue c;
        img->PALmap[i] = p;
```

---

kfax'buildMap() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1664)

```
buildMap(TIFFRGBAImage* img)
{
    switch (img->photometric) {
    case PHOTOMETRIC_RGB:
    case PHOTOMETRIC_YCBCR:
    case PHOTOMETRIC_SEPARATED:
        if (img->bitspersample == 8)
            break;
        /* fall thru... */
    case PHOTOMETRIC_MINISBLACK:
```

```
    case PHOTOMETRIC_MINISWHITE:
        if (!setupMap(img))
            return (0);
        break;
    case PHOTOMETRIC_PALETTE:
        /*
         * Convert 16-bit colormap to 8-bit (unless it looks
         * like an old-style 8-bit colormap).
         */
        if (checkcmap(img) == 16)
            cvtcmap(img);
        else
            TIFFWarning(TIFFFileName(img->tif), "Assuming 8-bit colormap");
        /*
         * Use mapping table and colormap to construct
         * unpacking tables for samples < 8 bits.
         */
        if (img->bitspersample <= 8 && !makecmap(img))
            return (0);
        break;
    }
    return (1);
}

/*
 * Select the appropriate conversion routine for packed data.
 */
static int
```

---

kfax'pickTileContigCase() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1702)

```
pickTileContigCase(TIFFRGBAImage* img)
{
    tileContigRoutine put = 0;

    if (buildMap(img)) {
        switch (img->photometric) {
        case PHOTOMETRIC_RGB:
            switch (img->bitspersample) {
            case 8:
                if (!img->Map) {
                    if (img->alpha == EXTRASAMPLE_ASSOCALPHA)
                        put = putRGBAAcontig8bittile;
                    else if (img->alpha == EXTRASAMPLE_UNASSALPHA)
                        put = putRGBUAcontig8bittile;
                    else
                        put = putRGBcontig8bittile;
                } else
                    put = putRGBcontig8bitMaptile;
                break;
            case 16:
                put = putRGBcontig16bittile;
                if (!img->Map) {
                    if (img->alpha == EXTRASAMPLE_ASSOCALPHA)
                        put = putRGBAAcontig16bittile;
                    else if (img->alpha == EXTRASAMPLE_UNASSALPHA)
                        put = putRGBUAcontig16bittile;
                }
```

```
                break;
            }
            break;
        case PHOTOMETRIC_SEPARATED:
            if (img->bitspersample == 8) {
                if (!img->Map)
                    put = putRGBcontig8bitCMYKtile;
                else
                    put = putRGBcontig8bitCMYKMaptile;
            }
            break;
        case PHOTOMETRIC_PALETTE:
            switch (img->bitspersample) {
            case 8:     put = put8bitcmaptile; break;
            case 4: put = put4bitcmaptile; break;
            case 2: put = put2bitcmaptile; break;
            case 1: put = put1bitcmaptile; break;
            }
            break;
        case PHOTOMETRIC_MINISWHITE:
        case PHOTOMETRIC_MINISBLACK:
            switch (img->bitspersample) {
            case 8:     put = putgreytile; break;
            case 4: put = put4bitbwtile; break;
            case 2: put = put2bitbwtile; break;
            case 1: put = put1bitbwtile; break;
            }
            break;
        case PHOTOMETRIC_YCBCR:
            if (img->bitspersample == 8)
                put = initYCbCrConversion(img);
            break;
        }
    }
    return ((img->put.contig = put) != 0);
}

/*
 * Select the appropriate conversion routine for unpacked data.
 *
 * NB: we assume that unpacked single channel data is directed
 *      to the "packed routines.
 */
static int
```

---

kfax'pickTileSeparateCase() (./kdegraphics/kfax/libtiffax/tif_getimage.c:1773)

```
pickTileSeparateCase(TIFFRGBAImage* img)
{
    tileSeparateRoutine put = 0;

    if (buildMap(img)) {
        switch (img->photometric) {
        case PHOTOMETRIC_RGB:
            switch (img->bitspersample) {
            case 8:
                if (!img->Map) {
                    if (img->alpha == EXTRASAMPLE_ASSOCALPHA)
```

```
                        put = putRGBAAseparate8bittile;
                    else if (img->alpha == EXTRASAMPLE_UNASSALPHA)
                        put = putRGBUAseparate8bittile;
                    else
                        put = putRGBseparate8bittile;
                } else
                    put = putRGBseparate8bitMaptile;
                break;
            case 16:
                put = putRGBseparate16bittile;
                if (!img->Map) {
                    if (img->alpha == EXTRASAMPLE_ASSOCALPHA)
                        put = putRGBAAseparate16bittile;
                    else if (img->alpha == EXTRASAMPLE_UNASSALPHA)
                        put = putRGBUAseparate16bittile;
                }
                break;
            }
            break;
        }
    }
    return ((img->put.separate = put) != 0);
}
```

---

kfax'TIFFjpeg_error_exit() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:148)

```
TIFFjpeg_error_exit(j_common_ptr cinfo)
{
        JPEGState *sp = (JPEGState *) cinfo;    /* NB: cinfo assumed first */
        char buffer[JMSG_LENGTH_MAX];

        (*cinfo->err->format_message) (cinfo, buffer);
        TIFFError("JPEGLib", buffer);           /* display the error message */
        jpeg_abort(cinfo);                      /* clean up libjpeg state */
        LONGJMP(sp->exit_jmpbuf, 1);            /* return to libtiff caller */
}

/*
 * This routine is invoked only for warning messages,
 * since error_exit does its own thing and trace_level
 * is never set > 0.
 */
static void
```

---

kfax'TIFFjpeg_output_message() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:165)

```
TIFFjpeg_output_message(j_common_ptr cinfo)
{
        char buffer[JMSG_LENGTH_MAX];

        (*cinfo->err->format_message) (cinfo, buffer);
        TIFFWarning("JPEGLib", buffer);
}

/*
 * Interface routines.  This layer of routines exists
```

```
 * primarily to limit side-effects from using setjmp.
 * Also, normal/error returns are converted into return
 * values per libtiff practice.
 */
```

---

kfax'TIFFjpeg_create_compress() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:183)

```
TIFFjpeg_create_compress(JPEGState* sp)
{
        /* initialize JPEG error handling */
        sp->cinfo.c.err = jpeg_std_error(&sp->err);
        sp->err.error_exit = TIFFjpeg_error_exit;
        sp->err.output_message = TIFFjpeg_output_message;

        return CALLVJPEG(sp, jpeg_create_compress(&sp->cinfo.c));
}

static int
```

---

kfax'TIFFjpeg_create_decompress() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:194)

```
TIFFjpeg_create_decompress(JPEGState* sp)
{
        /* initialize JPEG error handling */
        sp->cinfo.d.err = jpeg_std_error(&sp->err);
        sp->err.error_exit = TIFFjpeg_error_exit;
        sp->err.output_message = TIFFjpeg_output_message;

        return CALLVJPEG(sp, jpeg_create_decompress(&sp->cinfo.d));
}

static int
```

---

kfax'TIFFjpeg_set_defaults() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:205)

```
TIFFjpeg_set_defaults(JPEGState* sp)
{
        return CALLVJPEG(sp, jpeg_set_defaults(&sp->cinfo.c));
}

static int
```

---

kfax'TIFFjpeg_set_colorspace() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:211)

```
TIFFjpeg_set_colorspace(JPEGState* sp, J_COLOR_SPACE colorspace)
{
        return CALLVJPEG(sp, jpeg_set_colorspace(&sp->cinfo.c, colorspace));
}
```

```
static int
```

---

kfax'TIFFjpeg_set_quality() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:217)

```
TIFFjpeg_set_quality(JPEGState* sp, int quality, boolean force_baseline)
{
        return CALLVJPEG(sp,
            jpeg_set_quality(&sp->cinfo.c, quality, force_baseline));
}

static int
```

---

kfax'TIFFjpeg_suppress_tables() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:224)

```
TIFFjpeg_suppress_tables(JPEGState* sp, boolean suppress)
{
        return CALLVJPEG(sp, jpeg_suppress_tables(&sp->cinfo.c, suppress));
}

static int
```

---

kfax'TIFFjpeg_start_compress() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:230)

```
TIFFjpeg_start_compress(JPEGState* sp, boolean write_all_tables)
{
        return CALLVJPEG(sp,
            jpeg_start_compress(&sp->cinfo.c, write_all_tables));
}

static int
```

---

kfax'TIFFjpeg_write_scanlines() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:237)

```
TIFFjpeg_write_scanlines(JPEGState* sp, JSAMPARRAY scanlines, int num_lines)
{
        return CALLJPEG(sp, -1, (int) jpeg_write_scanlines(&sp->cinfo.c,
            scanlines, (JDIMENSION) num_lines));
}

static int
```

---

kfax'TIFFjpeg_write_raw_data() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:244)

```
TIFFjpeg_write_raw_data(JPEGState* sp, JSAMPIMAGE data, int num_lines)
{
        return CALLJPEG(sp, -1, (int) jpeg_write_raw_data(&sp->cinfo.c,
            data, (JDIMENSION) num_lines));
}
```

```
static int
```

---

kfax'TIFFjpeg_finish_compress() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:251)

```
TIFFjpeg_finish_compress(JPEGState* sp)
{
        return CALLVJPEG(sp, jpeg_finish_compress(&sp->cinfo.c));
}

static int
```

---

kfax'TIFFjpeg_write_tables() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:257)

```
TIFFjpeg_write_tables(JPEGState* sp)
{
        return CALLVJPEG(sp, jpeg_write_tables(&sp->cinfo.c));
}

static int
```

---

kfax'TIFFjpeg_read_header() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:263)

```
TIFFjpeg_read_header(JPEGState* sp, boolean require_image)
{
        return CALLJPEG(sp, -1, jpeg_read_header(&sp->cinfo.d, require_image));
}

static int
```

---

kfax'TIFFjpeg_start_decompress() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:269)

```
TIFFjpeg_start_decompress(JPEGState* sp)
{
        return CALLVJPEG(sp, jpeg_start_decompress(&sp->cinfo.d));
}

static int
```

---

kfax'TIFFjpeg_read_scanlines() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:275)

```
TIFFjpeg_read_scanlines(JPEGState* sp, JSAMPARRAY scanlines, int max_lines)
{
        return CALLJPEG(sp, -1, (int) jpeg_read_scanlines(&sp->cinfo.d,
            scanlines, (JDIMENSION) max_lines));
}

static int
```

---

kfax'TIFFjpeg_read_raw_data() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:282)

```
TIFFjpeg_read_raw_data(JPEGState* sp, JSAMPIMAGE data, int max_lines)
{
        return CALLJPEG(sp, -1, (int) jpeg_read_raw_data(&sp->cinfo.d,
            data, (JDIMENSION) max_lines));
}

static int
```

---

kfax'TIFFjpeg_finish_decompress() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:289)

```
TIFFjpeg_finish_decompress(JPEGState* sp)
{
        return CALLJPEG(sp, -1, (int) jpeg_finish_decompress(&sp->cinfo.d));
}

static int
```

---

kfax'TIFFjpeg_abort() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:295)

```
TIFFjpeg_abort(JPEGState* sp)
{
        return CALLVJPEG(sp, jpeg_abort(&sp->cinfo.comm));
}

static int
```

---

kfax'TIFFjpeg_destroy() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:301)

```
TIFFjpeg_destroy(JPEGState* sp)
{
        return CALLVJPEG(sp, jpeg_destroy(&sp->cinfo.comm));
}

static JSAMPARRAY
```

---

kfax'TIFFjpeg_alloc_sarray() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:307)

```
TIFFjpeg_alloc_sarray(JPEGState* sp, int pool_id,
                      JDIMENSION samplesperrow, JDIMENSION numrows)
{
        return CALLJPEG(sp, (JSAMPARRAY) NULL,
            (*sp->cinfo.comm.mem->alloc_sarray)
                (&sp->cinfo.comm, pool_id, samplesperrow, numrows));
}
```

```
/*
 * JPEG library destination data manager.
 * These routines direct compressed data from libjpeg into the
 * libtiff output buffer.
 */

static void
```

---

kfax'std_init_destination() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:322)

```
std_init_destination(j_compress_ptr cinfo)
{
        JPEGState* sp = (JPEGState*) cinfo;
        TIFF* tif = sp->tif;

        sp->dest.next_output_byte = (JOCTET*) tif->tif_rawdata;
        sp->dest.free_in_buffer = (size_t) tif->tif_rawdatasize;
}

static boolean
```

---

kfax'std_empty_output_buffer() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:332)

```
std_empty_output_buffer(j_compress_ptr cinfo)
{
        JPEGState* sp = (JPEGState*) cinfo;
        TIFF* tif = sp->tif;

        /* the entire buffer has been filled */
        tif->tif_rawcc = tif->tif_rawdatasize;
        TIFFFlushData1(tif);
        sp->dest.next_output_byte = (JOCTET*) tif->tif_rawdata;
        sp->dest.free_in_buffer = (size_t) tif->tif_rawdatasize;

        return (TRUE);
}

static void
```

---

kfax'std_term_destination() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:347)

```
std_term_destination(j_compress_ptr cinfo)
{
        JPEGState* sp = (JPEGState*) cinfo;
        TIFF* tif = sp->tif;

        tif->tif_rawcp = (tidata_t) sp->dest.next_output_byte;
        tif->tif_rawcc =
            tif->tif_rawdatasize - (tsize_t) sp->dest.free_in_buffer;
        /* NB: libtiff does the final buffer flush */
}

static void
```

---

kfax'TIFFjpeg_data_dest() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:359)

```
TIFFjpeg_data_dest(JPEGState* sp, TIFF* tif)
{
        (void) tif;
        sp->cinfo.c.dest = &sp->dest;
        sp->dest.init_destination = std_init_destination;
        sp->dest.empty_output_buffer = std_empty_output_buffer;
        sp->dest.term_destination = std_term_destination;
}

/*
 * Alternate destination manager for outputting to JPEGTables field.
 */

static void
```

---

kfax'tables_init_destination() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:373)

```
tables_init_destination(j_compress_ptr cinfo)
{
        JPEGState* sp = (JPEGState*) cinfo;

        /* while building, jpegtables_length is allocated buffer size */
        sp->dest.next_output_byte = (JOCTET*) sp->jpegtables;
        sp->dest.free_in_buffer = (size_t) sp->jpegtables_length;
}

static boolean
```

---

kfax'tables_empty_output_buffer() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:383)

```
tables_empty_output_buffer(j_compress_ptr cinfo)
{
        JPEGState* sp = (JPEGState*) cinfo;
        void* newbuf;

        /* the entire buffer has been filled; enlarge it by 1000 bytes */
        newbuf = _TIFFrealloc((tdata_t) sp->jpegtables,
                              (tsize_t) (sp->jpegtables_length + 1000));
        if (newbuf == NULL)
                ERREXIT1(cinfo, JERR_OUT_OF_MEMORY, 100);
        sp->dest.next_output_byte = (JOCTET*) newbuf + sp->jpegtables_length;
        sp->dest.free_in_buffer = (size_t) 1000;
        sp->jpegtables = newbuf;
        sp->jpegtables_length += 1000;
        return (TRUE);
}

static void
```

---

kfax'tables_term_destination() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:401)

```
tables_term_destination(j_compress_ptr cinfo)
{
        JPEGState* sp = (JPEGState*) cinfo;

        /* set tables length to number of bytes actually emitted */
        sp->jpegtables_length -= sp->dest.free_in_buffer;
}

static int
```

---

kfax'TIFFjpeg_tables_dest() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:410)

```
TIFFjpeg_tables_dest(JPEGState* sp, TIFF* tif)
{
        (void) tif;
        /*
         * Allocate a working buffer for building tables.
         * Initial size is 1000 bytes, which is usually adequate.
         */
        if (sp->jpegtables)
                _TIFFfree(sp->jpegtables);
        sp->jpegtables_length = 1000;
        sp->jpegtables = (void*) _TIFFmalloc((tsize_t) sp->jpegtables_length);
        if (sp->jpegtables == NULL) {
                sp->jpegtables_length = 0;
                TIFFError("TIFFjpeg_tables_dest", "No space for JPEGTables");
                return (0);
        }
        sp->cinfo.c.dest = &sp->dest;
        sp->dest.init_destination = tables_init_destination;
        sp->dest.empty_output_buffer = tables_empty_output_buffer;
        sp->dest.term_destination = tables_term_destination;
        return (1);
}

/*
 * JPEG library source data manager.
 * These routines supply compressed data to libjpeg.
 */

static void
```

---

kfax'std_init_source() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:439)

```
std_init_source(j_decompress_ptr cinfo)
{
        JPEGState* sp = (JPEGState*) cinfo;
        TIFF* tif = sp->tif;

        sp->src.next_input_byte = (const JOCTET*) tif->tif_rawdata;
        sp->src.bytes_in_buffer = (size_t) tif->tif_rawcc;
}
```

```
static boolean
```

---

kfax'std_fill_input_buffer() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:449)

```
std_fill_input_buffer(j_decompress_ptr cinfo)
{
	JPEGState* sp = (JPEGState* ) cinfo;
	static const JOCTET dummy_EOI[2] = { 0xFF, JPEG_EOI };

	/*
	 * Should never get here since entire strip/tile is
	 * read into memory before the decompressor is called,
	 * and thus was supplied by init_source.
	 */
	WARNMS(cinfo, JWRN_JPEG_EOF);
	/* insert a fake EOI marker */
	sp->src.next_input_byte = dummy_EOI;
	sp->src.bytes_in_buffer = 2;
	return (TRUE);
}

static void
```

---

kfax'std_skip_input_data() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:467)

```
std_skip_input_data(j_decompress_ptr cinfo, long num_bytes)
{
	JPEGState* sp = (JPEGState*) cinfo;

	if (num_bytes > 0) {
		if (num_bytes > (long) sp->src.bytes_in_buffer) {
			/* oops, buffer overrun */
			(void) std_fill_input_buffer(cinfo);
		} else {
			sp->src.next_input_byte += (size_t) num_bytes;
			sp->src.bytes_in_buffer -= (size_t) num_bytes;
		}
	}
}

static void
```

---

kfax'std_term_source() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:483)

```
std_term_source(j_decompress_ptr cinfo)
{
	/* No work necessary here */
	/* Or must we update tif->tif_rawcp, tif->tif_rawcc ??? */
	/* (if so, need empty tables_term_source!) */
	(void) cinfo;
}
```

```
static void
```

---

kfax'TIFFjpeg_data_src() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:492)

```
TIFFjpeg_data_src(JPEGState* sp, TIFF* tif)
{
        (void) tif;
        sp->cinfo.d.src = &sp->src;
        sp->src.init_source = std_init_source;
        sp->src.fill_input_buffer = std_fill_input_buffer;
        sp->src.skip_input_data = std_skip_input_data;
        sp->src.resync_to_restart = jpeg_resync_to_restart;
        sp->src.term_source = std_term_source;
        sp->src.bytes_in_buffer = 0;            /* for safety */
        sp->src.next_input_byte = NULL;
}

/*
 * Alternate source manager for reading from JPEGTables.
 * We can share all the code except for the init routine.
 */

static void
```

---

kfax'tables_init_source() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:511)

```
tables_init_source(j_decompress_ptr cinfo)
{
        JPEGState* sp = (JPEGState*) cinfo;

        sp->src.next_input_byte = (const JOCTET*) sp->jpegtables;
        sp->src.bytes_in_buffer = (size_t) sp->jpegtables_length;
}

static void
```

---

kfax'TIFFjpeg_tables_src() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:520)

```
TIFFjpeg_tables_src(JPEGState* sp, TIFF* tif)
{
        TIFFjpeg_data_src(sp, tif);
        sp->src.init_source = tables_init_source;
}

/*
 * Allocate downsampled-data buffers needed for downsampled I/O.
 * We use values computed in jpeg_start_compress or jpeg_start_decompress.
 * We use libjpeg's allocator so that buffers will be released automatically
 * when done with strip/tile.
 * This is also a handy place to compute samplesperclump, bytesperline.
 */
static int
```

kfax'alloc_downsampled_buffers() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:534)

```
alloc_downsampled_buffers(TIFF* tif, jpeg_component_info* comp_info,
                          int num_components)
{
        JPEGState* sp = JState(tif);
        int ci;
        jpeg_component_info* compptr;
        JSAMPARRAY buf;
        int samples_per_clump = 0;

        for (ci = 0, compptr = comp_info; ci < num_components;
             ci++, compptr++) {
                samples_per_clump += compptr->h_samp_factor *
                        compptr->v_samp_factor;
                buf = TIFFjpeg_alloc_sarray(sp, JPOOL_IMAGE,
                                compptr->width_in_blocks * DCTSIZE,
                                (JDIMENSION) (compptr->v_samp_factor*DCTSIZE));
                if (buf == NULL)
                        return (0);
                sp->ds_buffer[ci] = buf;
        }
        sp->samplesperclump = samples_per_clump;
        /* Cb,Cr both have sampling factors 1 */
        /* so downsampled width of Cb is # of clumps per line */
        sp->bytesperline = sizeof(JSAMPLE) * samples_per_clump *
                comp_info[1].downsampled_width;
        return (1);
}


/*
 * JPEG Decoding.
 */

static int
```

kfax'JPEGSetupDecode() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:568)

```
JPEGSetupDecode(TIFF* tif)
{
        JPEGState* sp = JState(tif);
        TIFFDirectory *td = &tif->tif_dir;

        assert(sp != NULL);
        assert(sp->cinfo.comm.is_decompressor);

        /* Read JPEGTables if it is present */
        if (TIFFFieldSet(tif,FIELD_JPEGTABLES)) {
                TIFFjpeg_tables_src(sp, tif);
                if(TIFFjpeg_read_header(sp,FALSE) != JPEG_HEADER_TABLES_ONLY) {
                        TIFFError("JPEGSetupDecode", "Bogus JPEGTables field");
                        return (0);
                }
        }
```

```
        /* Grab parameters that are same for all strips/tiles */
        sp->photometric = td->td_photometric;
        switch (sp->photometric) {
        case PHOTOMETRIC_YCBCR:
                sp->h_sampling = td->td_ycbcrsubsampling[0];
                sp->v_sampling = td->td_ycbcrsubsampling[1];
                break;
        default:
                /* TIFF 6.0 forbids subsampling of all other color spaces */
                sp->h_sampling = 1;
                sp->v_sampling = 1;
                break;
        }

        /* Set up for reading normal data */
        TIFFjpeg_data_src(sp, tif);
        tif->tif_postdecode = _TIFFNoPostDecode; /* override byte swapping */
        return (1);
}

/*
 * Set up for decoding a strip or tile.
 */
static int
```

---

kfax'JPEGPreDecode() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:609)

```
JPEGPreDecode(TIFF* tif, tsample_t s)
{
        JPEGState *sp = JState(tif);
        TIFFDirectory *td = &tif->tif_dir;
        static char module[] = "JPEGPreDecode";
        uint32 segment_width, segment_height;
        int downsampled_output;
        int ci;

        assert(sp != NULL);
        assert(sp->cinfo.comm.is_decompressor);
        /*
         * Reset decoder state from any previous strip/tile,
         * in case application didn't read the whole strip.
         */
        if (!TIFFjpeg_abort(sp))
                return (0);
        /*
         * Read the header for this strip/tile.
         */
        if (TIFFjpeg_read_header(sp, TRUE) != JPEG_HEADER_OK)
                return (0);
        /*
         * Check image parameters and set decompression parameters.
         */
        if (isTiled(tif)) {
                segment_width = td->td_tilewidth;
                segment_height = td->td_tilelength;
                sp->bytesperline = TIFFTileRowSize(tif);
        } else {
```

```
                segment_width = td->td_imagewidth;
                segment_height = td->td_imagelength - tif->tif_row;
                if (segment_height > td->td_rowsperstrip)
                        segment_height = td->td_rowsperstrip;
                sp->bytesperline = TIFFScanlineSize(tif);
        }
        if (td->td_planarconfig == PLANARCONFIG_SEPARATE && s > 0) {
                /*
                 * For PC 2, scale down the expected strip/tile size
                 * to match a downsampled component
                 */
                segment_width = TIFFhowmany(segment_width, sp->h_sampling);
                segment_height = TIFFhowmany(segment_height, sp->v_sampling);
        }
        if (sp->cinfo.d.image_width != segment_width ||
            sp->cinfo.d.image_height != segment_height) {
                TIFFError(module, "Improper JPEG strip/tile size");
                return (0);
        }
        if (sp->cinfo.d.num_components !=
            (td->td_planarconfig == PLANARCONFIG_CONTIG ?
             td->td_samplesperpixel : 1)) {
                TIFFError(module, "Improper JPEG component count");
                return (0);
        }
        if (sp->cinfo.d.data_precision != td->td_bitspersample) {
                TIFFError(module, "Improper JPEG data precision");
                return (0);
        }
        if (td->td_planarconfig == PLANARCONFIG_CONTIG) {
                /* Component 0 should have expected sampling factors */
                if (sp->cinfo.d.comp_info[0].h_samp_factor != sp->h_sampling ||
                    sp->cinfo.d.comp_info[0].v_samp_factor != sp->v_sampling) {
                        TIFFError(module, "Improper JPEG sampling factors");
                        return (0);
                }
                /* Rest should have sampling factors 1,1 */
                for (ci = 1; ci < sp->cinfo.d.num_components; ci++) {
                        if (sp->cinfo.d.comp_info[ci].h_samp_factor != 1 ||
                            sp->cinfo.d.comp_info[ci].v_samp_factor != 1) {
                                TIFFError(module, "Improper JPEG sampling factor
                                return (0);
                        }
                }
        } else {
                /* PC 2's single component should have sampling factors 1,1 */
                if (sp->cinfo.d.comp_info[0].h_samp_factor != 1 ||
                    sp->cinfo.d.comp_info[0].v_samp_factor != 1) {
                        TIFFError(module, "Improper JPEG sampling factors");
                        return (0);
                }
        }
        downsampled_output = FALSE;
        if (td->td_planarconfig == PLANARCONFIG_CONTIG &&
            sp->photometric == PHOTOMETRIC_YCBCR &&
            sp->jpegcolormode == JPEGCOLORMODE_RGB) {
                /* Convert YCbCr to RGB */
                sp->cinfo.d.jpeg_color_space = JCS_YCbCr;
                sp->cinfo.d.out_color_space = JCS_RGB;
        } else {
                /* Suppress colorspace handling */
```

```
                sp->cinfo.d.jpeg_color_space = JCS_UNKNOWN;
                sp->cinfo.d.out_color_space = JCS_UNKNOWN;
                if (td->td_planarconfig == PLANARCONFIG_CONTIG &&
                    (sp->h_sampling != 1 || sp->v_sampling != 1))
                        downsampled_output = TRUE;
                /* XXX what about up-sampling? */
        }
        if (downsampled_output) {
                /* Need to use raw-data interface to libjpeg */
                sp->cinfo.d.raw_data_out = TRUE;
                tif->tif_decoderow = JPEGDecodeRaw;
                tif->tif_decodestrip = JPEGDecodeRaw;
                tif->tif_decodetile = JPEGDecodeRaw;
        } else {
                /* Use normal interface to libjpeg */
                sp->cinfo.d.raw_data_out = FALSE;
                tif->tif_decoderow = JPEGDecode;
                tif->tif_decodestrip = JPEGDecode;
                tif->tif_decodetile = JPEGDecode;
        }
        /* Start JPEG decompressor */
        if (!TIFFjpeg_start_decompress(sp))
                return (0);
        /* Allocate downsampled-data buffers if needed */
        if (downsampled_output) {
                if (!alloc_downsampled_buffers(tif, sp->cinfo.d.comp_info,
                                               sp->cinfo.d.num_components))
                        return (0);
                sp->scancount = DCTSIZE;        /* mark buffer empty */
        }
        return (1);
}

/*
 * Decode a chunk of pixels.
 * "Standard" case: returned data is not downsampled.
 */
static int
```

---

kfax'JPEGDecode() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:738)

```
JPEGDecode(TIFF* tif, tidata_t buf, tsize_t cc, tsample_t s)
{
        JPEGState *sp = JState(tif);
        tsize_t nrows;
        JSAMPROW bufptr[1];

        (void) s;
        assert(sp != NULL);
        /* data is expected to be read in multiples of a scanline */
        nrows = cc / sp->bytesperline;
        if (cc % sp->bytesperline)
                TIFFWarning(tif->tif_name, "fractional scanline not read");

        while (nrows-- > 0) {
                bufptr[0] = (JSAMPROW) buf;
                if (TIFFjpeg_read_scanlines(sp, bufptr, 1) != 1)
                        return (0);
```

```
                if (nrows > 0)
                        tif->tif_row++;
                buf += sp->bytesperline;
        }
        /* Close down the decompressor if we've finished the strip or tile. */
        if (sp->cinfo.d.output_scanline == sp->cinfo.d.output_height) {
                if (TIFFjpeg_finish_decompress(sp) != TRUE)
                        return (0);
        }
        return (1);
}

/*
 * Decode a chunk of pixels.
 * Returned data is downsampled per sampling factors.
 */
static int
```

---

kfax'JPEGDecodeRaw() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:772)

```
JPEGDecodeRaw(TIFF* tif, tidata_t buf, tsize_t cc, tsample_t s)
{
        JPEGState *sp = JState(tif);
        JSAMPLE* inptr;
        JSAMPLE* outptr;
        tsize_t nrows;
        JDIMENSION clumps_per_line, nclump;
        int clumpoffset, ci, xpos, ypos;
        jpeg_component_info* compptr;
        int samples_per_clump = sp->samplesperclump;

        (void) s;
        assert(sp != NULL);
        /* data is expected to be read in multiples of a scanline */
        nrows = cc / sp->bytesperline;
        if (cc % sp->bytesperline)
                TIFFWarning(tif->tif_name, "fractional scanline not read");

        /* Cb,Cr both have sampling factors 1, so this is correct */
        clumps_per_line = sp->cinfo.d.comp_info[1].downsampled_width;

        while (nrows-- > 0) {
                /* Reload downsampled-data buffer if needed */
                if (sp->scancount >= DCTSIZE) {
                        int n = sp->cinfo.d.max_v_samp_factor * DCTSIZE;
                        if (TIFFjpeg_read_raw_data(sp, sp->ds_buffer, n) != n)
                                return (0);
                        sp->scancount = 0;
                        /* Close down the decompressor if done. */
                        if (sp->cinfo.d.output_scanline >=
                            sp->cinfo.d.output_height) {
                                if (TIFFjpeg_finish_decompress(sp) != TRUE)
                                        return (0);
                        }
                }
                /*
                 * Fastest way to unseparate the data is to make one pass
                 * over the scanline for each row of each component.
```

```
                 */
                clumpoffset = 0;                /* first sample in clump */
                for (ci = 0, compptr = sp->cinfo.d.comp_info;
                     ci < sp->cinfo.d.num_components;
                     ci++, compptr++) {
                    int hsamp = compptr->h_samp_factor;
                    int vsamp = compptr->v_samp_factor;
                    for (ypos = 0; ypos < vsamp; ypos++) {
                        inptr = sp->ds_buffer[ci][sp->scancount*vsamp + ypos];
                        outptr = ((JSAMPLE*) buf) + clumpoffset;
                        if (hsamp == 1) {
                            /* fast path for at least Cb and Cr */
                            for (nclump = clumps_per_line; nclump-- > 0; ) {
                                outptr[0] = *inptr++;
                                outptr += samples_per_clump;
                            }
                        } else {
                            /* general case */
                            for (nclump = clumps_per_line; nclump-- > 0; ) {
                                for (xpos = 0; xpos < hsamp; xpos++)
                                    outptr[xpos] = *inptr++;
                                outptr += samples_per_clump;
                            }
                        }
                        clumpoffset += hsamp;
                    }
                }
                sp->scancount++;
                if (nrows > 0)
                        tif->tif_row++;
                buf += sp->bytesperline;
        }
        return (1);
}


/*
 * JPEG Encoding.
 */

static void
```

---

kfax'unsuppress_quant_table() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:851)

```
unsuppress_quant_table (JPEGState* sp, int tblno)
{
        JQUANT_TBL* qtbl;

        if ((qtbl = sp->cinfo.c.quant_tbl_ptrs[tblno]) != NULL)
                qtbl->sent_table = FALSE;
}

static void
```

---

kfax'unsuppress_huff_table() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:860)

```
unsuppress_huff_table (JPEGState* sp, int tblno)
{
	JHUFF_TBL* htbl;

	if ((htbl = sp->cinfo.c.dc_huff_tbl_ptrs[tblno]) != NULL)
		htbl->sent_table = FALSE;
	if ((htbl = sp->cinfo.c.ac_huff_tbl_ptrs[tblno]) != NULL)
		htbl->sent_table = FALSE;
}

static int
```

---

kfax'prepare_JPEGTables() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:871)

```
prepare_JPEGTables(TIFF* tif)
{
	JPEGState* sp = JState(tif);

	/* Initialize quant tables for current quality setting */
	if (!TIFFjpeg_set_quality(sp, sp->jpegquality, FALSE))
		return (0);
	/* Mark only the tables we want for output */
	/* NB: chrominance tables are currently used only with YCbCr */
	if (!TIFFjpeg_suppress_tables(sp, TRUE))
		return (0);
	if (sp->jpegtablesmode & JPEGTABLESMODE_QUANT) {
		unsuppress_quant_table(sp, 0);
		if (sp->photometric == PHOTOMETRIC_YCBCR)
			unsuppress_quant_table(sp, 1);
	}
	if (sp->jpegtablesmode & JPEGTABLESMODE_HUFF) {
		unsuppress_huff_table(sp, 0);
		if (sp->photometric == PHOTOMETRIC_YCBCR)
			unsuppress_huff_table(sp, 1);
	}
	/* Direct libjpeg output into jpegtables */
	if (!TIFFjpeg_tables_dest(sp, tif))
		return (0);
	/* Emit tables-only datastream */
	if (!TIFFjpeg_write_tables(sp))
		return (0);

	return (1);
}

static int
```

---

kfax'JPEGSetupEncode() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:903)

```
JPEGSetupEncode(TIFF* tif)
{
	JPEGState* sp = JState(tif);
	TIFFDirectory *td = &tif->tif_dir;
	static char module[] = "JPEGSetupEncode";
```

```
        assert(sp != NULL);
        assert(!sp->cinfo.comm.is_decompressor);

        /*
         * Initialize all JPEG parameters to default values.
         * Note that jpeg_set_defaults needs legal values for
         * in_color_space and input_components.
         */
        sp->cinfo.c.in_color_space = JCS_UNKNOWN;
        sp->cinfo.c.input_components = 1;
        if (!TIFFjpeg_set_defaults(sp))
                return (0);
        /* Set per-file parameters */
        sp->photometric = td->td_photometric;
        switch (sp->photometric) {
        case PHOTOMETRIC_YCBCR:
                sp->h_sampling = td->td_ycbcrsubsampling[0];
                sp->v_sampling = td->td_ycbcrsubsampling[1];
                /*
                 * A ReferenceBlackWhite field *must* be present since the
                 * default value is inappropriate for YCbCr.  Fill in the
                 * proper value if application didn't set it.
                 */
#ifdef COLORIMETRY_SUPPORT
                if (!TIFFFieldSet(tif, FIELD_REFBLACKWHITE)) {
                        float refbw[6];
                        long top = 1L << td->td_bitspersample;
                        refbw[0] = 0;
                        refbw[1] = (float)(top-1L);
                        refbw[2] = (float)(top>>1);
                        refbw[3] = refbw[1];
                        refbw[4] = refbw[2];
                        refbw[5] = refbw[1];
                        TIFFSetField(tif, TIFFTAG_REFERENCEBLACKWHITE, refbw);
                }
#endif
                break;
        case PHOTOMETRIC_PALETTE:               /* disallowed by Tech Note */
        case PHOTOMETRIC_MASK:
                TIFFError(module,
                          "PhotometricInterpretation %d not allowed for JPEG",
                          (int) sp->photometric);
                return (0);
        default:
                /* TIFF 6.0 forbids subsampling of all other color spaces */
                sp->h_sampling = 1;
                sp->v_sampling = 1;
                break;
        }

        /* Verify miscellaneous parameters */

        /*
         * This would need work if libtiff ever supports different
         * depths for different components, or if libjpeg ever supports
         * run-time selection of depth.  Neither is imminent.
         */
        if (td->td_bitspersample != BITS_IN_JSAMPLE) {
                TIFFError(module, "BitsPerSample %d not allowed for JPEG",
                          (int) td->td_bitspersample);
                return (0);
```

```
        }
        sp->cinfo.c.data_precision = td->td_bitspersample;
        if (isTiled(tif)) {
                if ((td->td_tilelength % (sp->v_sampling * DCTSIZE)) != 0) {
                        TIFFError(module,
                                  "JPEG tile height must be multiple of %d",
                                  sp->v_sampling * DCTSIZE);
                        return (0);
                }
                if ((td->td_tilewidth % (sp->h_sampling * DCTSIZE)) != 0) {
                        TIFFError(module,
                                  "JPEG tile width must be multiple of %d",
                                  sp->h_sampling * DCTSIZE);
                        return (0);
                }
        } else {
                if (td->td_rowsperstrip < td->td_imagelength &&
                    (td->td_rowsperstrip % (sp->v_sampling * DCTSIZE)) != 0) {
                        TIFFError(module,
                                  "RowsPerStrip must be multiple of %d for JPEG"
                                  sp->v_sampling * DCTSIZE);
                        return (0);
                }
        }

        /* Create a JPEGTables field if appropriate */
        if (sp->jpegtablesmode & (JPEGTABLESMODE_QUANT|JPEGTABLESMODE_HUFF)) {
                if (!prepare_JPEGTables(tif))
                        return (0);
                /* Mark the field present */
                /* Can't use TIFFSetField since BEENWRITING is already set! */
                TIFFSetFieldBit(tif, FIELD_JPEGTABLES);
                tif->tif_flags |= TIFF_DIRTYDIRECT;
        } else {
                /* We do not support application-supplied JPEGTables, */
                /* so mark the field not present */
                TIFFClrFieldBit(tif, FIELD_JPEGTABLES);
        }

        /* Direct libjpeg output to libtiff's output buffer */
        TIFFjpeg_data_dest(sp, tif);

        return (1);
}

/*
 * Set encoding state at the start of a strip or tile.
 */
static int
```

---

kfax'JPEGPreEncode() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1019)

```
JPEGPreEncode(TIFF* tif, tsample_t s)
{
        JPEGState *sp = JState(tif);
        TIFFDirectory *td = &tif->tif_dir;
        static char module[] = "JPEGPreEncode";
        uint32 segment_width, segment_height;
```

```
	int downsampled_input;

	assert(sp != NULL);
	assert(!sp->cinfo.comm.is_decompressor);
	/*
	 * Set encoding parameters for this strip/tile.
	 */
	if (isTiled(tif)) {
		segment_width = td->td_tilewidth;
		segment_height = td->td_tilelength;
		sp->bytesperline = TIFFTileRowSize(tif);
	} else {
		segment_width = td->td_imagewidth;
		segment_height = td->td_imagelength - tif->tif_row;
		if (segment_height > td->td_rowsperstrip)
			segment_height = td->td_rowsperstrip;
		sp->bytesperline = TIFFScanlineSize(tif);
	}
	if (td->td_planarconfig == PLANARCONFIG_SEPARATE && s > 0) {
		/* for PC 2, scale down the strip/tile size
		 * to match a downsampled component
		 */
		segment_width = TIFFhowmany(segment_width, sp->h_sampling);
		segment_height = TIFFhowmany(segment_height, sp->v_sampling);
	}
	if (segment_width > 65535 || segment_height > 65535) {
		TIFFError(module, "Strip/tile too large for JPEG");
		return (0);
	}
	sp->cinfo.c.image_width = segment_width;
	sp->cinfo.c.image_height = segment_height;
	downsampled_input = FALSE;
	if (td->td_planarconfig == PLANARCONFIG_CONTIG) {
		sp->cinfo.c.input_components = td->td_samplesperpixel;
		if (sp->photometric == PHOTOMETRIC_YCBCR) {
			if (sp->jpegcolormode == JPEGCOLORMODE_RGB) {
				sp->cinfo.c.in_color_space = JCS_RGB;
			} else {
				sp->cinfo.c.in_color_space = JCS_YCbCr;
				if (sp->h_sampling != 1 || sp->v_sampling != 1)
					downsampled_input = TRUE;
			}
			if (!TIFFjpeg_set_colorspace(sp, JCS_YCbCr))
				return (0);
			/*
			 * Set Y sampling factors;
			 * we assume jpeg_set_colorspace() set the rest to 1
			 */
			sp->cinfo.c.comp_info[0].h_samp_factor = sp->h_sampling;
			sp->cinfo.c.comp_info[0].v_samp_factor = sp->v_sampling;
		} else {
			sp->cinfo.c.in_color_space = JCS_UNKNOWN;
			if (!TIFFjpeg_set_colorspace(sp, JCS_UNKNOWN))
				return (0);
			/* jpeg_set_colorspace set all sampling factors to 1 */
		}
	} else {
		sp->cinfo.c.input_components = 1;
		sp->cinfo.c.in_color_space = JCS_UNKNOWN;
		if (!TIFFjpeg_set_colorspace(sp, JCS_UNKNOWN))
			return (0);
```

```
                sp->cinfo.c.comp_info[0].component_id = s;
                /* jpeg_set_colorspace() set sampling factors to 1 */
                if (sp->photometric == PHOTOMETRIC_YCBCR && s > 0) {
                        sp->cinfo.c.comp_info[0].quant_tbl_no = 1;
                        sp->cinfo.c.comp_info[0].dc_tbl_no = 1;
                        sp->cinfo.c.comp_info[0].ac_tbl_no = 1;
                }
        }
        /* ensure libjpeg won't write any extraneous markers */
        sp->cinfo.c.write_JFIF_header = FALSE;
        sp->cinfo.c.write_Adobe_marker = FALSE;
        /* set up table handling correctly */
        if (! (sp->jpegtablesmode & JPEGTABLESMODE_QUANT)) {
                if (!TIFFjpeg_set_quality(sp, sp->jpegquality, FALSE))
                        return (0);
                unsuppress_quant_table(sp, 0);
                unsuppress_quant_table(sp, 1);
        }
        if (sp->jpegtablesmode & JPEGTABLESMODE_HUFF)
                sp->cinfo.c.optimize_coding = FALSE;
        else
                sp->cinfo.c.optimize_coding = TRUE;
        if (downsampled_input) {
                /* Need to use raw-data interface to libjpeg */
                sp->cinfo.c.raw_data_in = TRUE;
                tif->tif_encoderow = JPEGEncodeRaw;
                tif->tif_encodestrip = JPEGEncodeRaw;
                tif->tif_encodetile = JPEGEncodeRaw;
        } else {
                /* Use normal interface to libjpeg */
                sp->cinfo.c.raw_data_in = FALSE;
                tif->tif_encoderow = JPEGEncode;
                tif->tif_encodestrip = JPEGEncode;
                tif->tif_encodetile = JPEGEncode;
        }
        /* Start JPEG compressor */
        if (!TIFFjpeg_start_compress(sp, FALSE))
                return (0);
        /* Allocate downsampled-data buffers if needed */
        if (downsampled_input) {
                if (!alloc_downsampled_buffers(tif, sp->cinfo.c.comp_info,
                                               sp->cinfo.c.num_components))
                        return (0);
        }
        sp->scancount = 0;

        return (1);
}

/*
 * Encode a chunk of pixels.
 * "Standard" case: incoming data is not downsampled.
 */
static int
```

---

kfax'JPEGEncode() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1140)

```
JPEGEncode(TIFF* tif, tidata_t buf, tsize_t cc, tsample_t s)
```

```
{
        JPEGState *sp = JState(tif);
        tsize_t nrows;
        JSAMPROW bufptr[1];

        (void) s;
        assert(sp != NULL);
        /* data is expected to be supplied in multiples of a scanline */
        nrows = cc / sp->bytesperline;
        if (cc % sp->bytesperline)
                TIFFWarning(tif->tif_name, "fractional scanline discarded");

        while (nrows-- > 0) {
                bufptr[0] = (JSAMPROW) buf;
                if (TIFFjpeg_write_scanlines(sp, bufptr, 1) != 1)
                        return (0);
                if (nrows > 0)
                        tif->tif_row++;
                buf += sp->bytesperline;
        }
        return (1);
}

/*
 * Encode a chunk of pixels.
 * Incoming data is expected to be downsampled per sampling factors.
 */
static int
```

---

kfax'JPEGEncodeRaw() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1169)

```
JPEGEncodeRaw(TIFF* tif, tidata_t buf, tsize_t cc, tsample_t s)
{
        JPEGState *sp = JState(tif);
        JSAMPLE* inptr;
        JSAMPLE* outptr;
        tsize_t nrows;
        JDIMENSION clumps_per_line, nclump;
        int clumpoffset, ci, xpos, ypos;
        jpeg_component_info* compptr;
        int samples_per_clump = sp->samplesperclump;

        (void) s;
        assert(sp != NULL);
        /* data is expected to be supplied in multiples of a scanline */
        nrows = cc / sp->bytesperline;
        if (cc % sp->bytesperline)
                TIFFWarning(tif->tif_name, "fractional scanline discarded");

        /* Cb,Cr both have sampling factors 1, so this is correct */
        clumps_per_line = sp->cinfo.c.comp_info[1].downsampled_width;

        while (nrows-- > 0) {
                /*
                 * Fastest way to separate the data is to make one pass
                 * over the scanline for each row of each component.
                 */
                clumpoffset = 0;                /* first sample in clump */
```

```
                for (ci = 0, compptr = sp->cinfo.c.comp_info;
                     ci < sp->cinfo.c.num_components;
                     ci++, compptr++) {
                    int hsamp = compptr->h_samp_factor;
                    int vsamp = compptr->v_samp_factor;
                    int padding = (int) (compptr->width_in_blocks * DCTSIZE -
                                         clumps_per_line * hsamp);
                    for (ypos = 0; ypos < vsamp; ypos++) {
                        inptr = ((JSAMPLE*) buf) + clumpoffset;
                        outptr = sp->ds_buffer[ci][sp->scancount*vsamp + ypos];
                        if (hsamp == 1) {
                            /* fast path for at least Cb and Cr */
                            for (nclump = clumps_per_line; nclump-- > 0; ) {
                                *outptr++ = inptr[0];
                                inptr += samples_per_clump;
                            }
                        } else {
                            /* general case */
                            for (nclump = clumps_per_line; nclump-- > 0; ) {
                                for (xpos = 0; xpos < hsamp; xpos++)
                                    *outptr++ = inptr[xpos];
                                inptr += samples_per_clump;
                            }
                        }
                        /* pad each scanline as needed */
                        for (xpos = 0; xpos < padding; xpos++) {
                            *outptr = outptr[-1];
                            outptr++;
                        }
                        clumpoffset += hsamp;
                    }
                }
                sp->scancount++;
                if (sp->scancount >= DCTSIZE) {
                        int n = sp->cinfo.c.max_v_samp_factor * DCTSIZE;
                        if (TIFFjpeg_write_raw_data(sp, sp->ds_buffer, n) != n)
                                return (0);
                        sp->scancount = 0;
                }
                if (nrows > 0)
                        tif->tif_row++;
                buf += sp->bytesperline;
        }
        return (1);
}

/*
 * Finish up at the end of a strip or tile.
 */
static int
```

---

kfax'JPEGPostEncode() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1246)

```
JPEGPostEncode(TIFF* tif)
{
        JPEGState *sp = JState(tif);

        if (sp->scancount > 0) {
```

```
                /*
                 * Need to emit a partial bufferload of downsampled data.
                 * Pad the data vertically.
                 */
                int ci, ypos, n;
                jpeg_component_info* compptr;

                for (ci = 0, compptr = sp->cinfo.c.comp_info;
                     ci < sp->cinfo.c.num_components;
                     ci++, compptr++) {
                        int vsamp = compptr->v_samp_factor;
                        tsize_t row_width = compptr->width_in_blocks * DCTSIZE
                                * sizeof(JSAMPLE);
                        for (ypos = sp->scancount * vsamp;
                             ypos < DCTSIZE * vsamp; ypos++) {
                                _TIFFmemcpy((tdata_t)sp->ds_buffer[ci][ypos],
                                            (tdata_t)sp->ds_buffer[ci][ypos-1],
                                            row_width);

                        }
                }
                n = sp->cinfo.c.max_v_samp_factor * DCTSIZE;
                if (TIFFjpeg_write_raw_data(sp, sp->ds_buffer, n) != n)
                        return (0);
        }

        return (TIFFjpeg_finish_compress(JState(tif)));
}

static void
```

---

kfax'JPEGCleanup() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1281)

```
JPEGCleanup(TIFF* tif)
{
        if (tif->tif_data) {
                JPEGState *sp = JState(tif);
                TIFFjpeg_destroy(sp);           /* release libjpeg resources */
                if (sp->jpegtables)             /* tag value */
                        _TIFFfree(sp->jpegtables);
                _TIFFfree(tif->tif_data);       /* release local state */
                tif->tif_data = NULL;
        }
}

static int
```

---

kfax'JPEGVSetField() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1294)

```
JPEGVSetField(TIFF* tif, ttag_t tag, va_list ap)
{
        JPEGState* sp = JState(tif);
        TIFFDirectory* td = &tif->tif_dir;
        uint32 v32;

        switch (tag) {
```

```
        case TIFFTAG_JPEGTABLES:
                v32 = va_arg(ap, uint32);
                if (v32 == 0) {
                        /* XXX */
                        return (0);
                }
                _TIFFsetByteArray(&sp->jpegtables, va_arg(ap, void*),
                    (long) v32);
                sp->jpegtables_length = v32;
                TIFFSetFieldBit(tif, FIELD_JPEGTABLES);
                break;
        case TIFFTAG_JPEGQUALITY:
                sp->jpegquality = va_arg(ap, int);
                return (1);                     /* pseudo tag */
        case TIFFTAG_JPEGCOLORMODE:
                sp->jpegcolormode = va_arg(ap, int);
                /*
                 * Mark whether returned data is up-sampled or not
                 * so TIFFStripSize and TIFFTileSize return values
                 * that reflect the true amount of data.
                 */
                tif->tif_flags &= ~TIFF_UPSAMPLED;
                if (td->td_planarconfig == PLANARCONFIG_CONTIG) {
                    if (td->td_photometric == PHOTOMETRIC_YCBCR &&
                      sp->jpegcolormode == JPEGCOLORMODE_RGB) {
                        tif->tif_flags |= TIFF_UPSAMPLED;
                    } else {
                        if (td->td_ycbcrsubsampling[0] != 1 ||
                            td->td_ycbcrsubsampling[1] != 1)
                            ; /* XXX what about up-sampling? */
                    }
                }
                /*
                 * Must recalculate cached tile size
                 * in case sampling state changed.
                 */
                tif->tif_tilesize = TIFFTileSize(tif);
                return (1);                     /* pseudo tag */
        case TIFFTAG_JPEGTABLESMODE:
                sp->jpegtablesmode = va_arg(ap, int);
                return (1);                     /* pseudo tag */
        default:
                return (*sp->vsetparent)(tif, tag, ap);
        }
        tif->tif_flags |= TIFF_DIRTYDIRECT;
        return (1);
}

static int
```

---

kfax'JPEGVGetField() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1350)

```
JPEGVGetField(TIFF* tif, ttag_t tag, va_list ap)
{
        JPEGState* sp = JState(tif);

        switch (tag) {
        case TIFFTAG_JPEGTABLES:
```

```
                /* u_short is bogus --- should be uint32 ??? */
                /* TIFFWriteNormalTag needs fixed  XXX */
                *va_arg(ap, u_short*) = (u_short) sp->jpegtables_length;
                *va_arg(ap, void**) = sp->jpegtables;
                break;
        case TIFFTAG_JPEGQUALITY:
                *va_arg(ap, int*) = sp->jpegquality;
                break;
        case TIFFTAG_JPEGCOLORMODE:
                *va_arg(ap, int*) = sp->jpegcolormode;
                break;
        case TIFFTAG_JPEGTABLESMODE:
                *va_arg(ap, int*) = sp->jpegtablesmode;
                break;
        default:
                return (*sp->vgetparent)(tif, tag, ap);
        }
        return (1);
}

static void
```

---

kfax'JPEGPrintDir() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1377)

```
JPEGPrintDir(TIFF* tif, FILE* fd, long flags)
{
        JPEGState* sp = JState(tif);

        (void) flags;
        if (TIFFFieldSet(tif,FIELD_JPEGTABLES))
                fprintf(fd, "  JPEG Tables: (%lu bytes)\n",
                        (u_long) sp->jpegtables_length);
}

static uint32
```

---

kfax'JPEGDefaultStripSize() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1388)

```
JPEGDefaultStripSize(TIFF* tif, uint32 s)
{
        JPEGState* sp = JState(tif);
        TIFFDirectory *td = &tif->tif_dir;

        s = (*sp->defsparent)(tif, s);
        if (s < td->td_imagelength)
                s = TIFFroundup(s, td->td_ycbcrsubsampling[1] * DCTSIZE);
        return (s);
}

static void
```

---

kfax'JPEGDefaultTileSize() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1400)

```
JPEGDefaultTileSize(TIFF* tif, uint32* tw, uint32* th)
{
        JPEGState* sp = JState(tif);
        TIFFDirectory *td = &tif->tif_dir;

        (*sp->deftparent)(tif, tw, th);
        *tw = TIFFroundup(*tw, td->td_ycbcrsubsampling[0] * DCTSIZE);
        *th = TIFFroundup(*th, td->td_ycbcrsubsampling[1] * DCTSIZE);
}

int
```

---

kfax'TIFFInitJPEG() (./kdegraphics/kfax/libtiffax/tif_jpeg.c:1411)

```
TIFFInitJPEG(TIFF* tif, int scheme)
{
        JPEGState* sp;

        assert(scheme == COMPRESSION_JPEG);

        /*
         * Allocate state block so tag methods have storage to record values.
         */
        tif->tif_data = (tidata_t) _TIFFmalloc(sizeof (JPEGState));
        if (tif->tif_data == NULL) {
                TIFFError("TIFFInitJPEG", "No space for JPEG state block");
                return (0);
        }
        sp = JState(tif);
        sp->tif = tif;                          /* back link */

        /*
         * Merge codec-specific tag information and
         * override parent get/set field methods.
         */
        _TIFFMergeFieldInfo(tif, jpegFieldInfo, N(jpegFieldInfo));
        sp->vgetparent = tif->tif_vgetfield;
        tif->tif_vgetfield = JPEGVGetField;     /* hook for codec tags */
        sp->vsetparent = tif->tif_vsetfield;
        tif->tif_vsetfield = JPEGVSetField;     /* hook for codec tags */
        tif->tif_printdir = JPEGPrintDir;       /* hook for codec tags */

        /* Default values for codec-specific fields */
        sp->jpegtables = NULL;
        sp->jpegtables_length = 0;
        sp->jpegquality = 75;                   /* Default IJG quality */
        sp->jpegcolormode = JPEGCOLORMODE_RAW;
        sp->jpegtablesmode = JPEGTABLESMODE_QUANT | JPEGTABLESMODE_HUFF;

        /*
         * Install codec methods.
         */
        tif->tif_setupdecode = JPEGSetupDecode;
        tif->tif_predecode = JPEGPreDecode;
        tif->tif_decoderow = JPEGDecode;
        tif->tif_decodestrip = JPEGDecode;
        tif->tif_decodetile = JPEGDecode;
```

```
        tif->tif_setupencode = JPEGSetupEncode;
        tif->tif_preencode = JPEGPreEncode;
        tif->tif_postencode = JPEGPostEncode;
        tif->tif_encoderow = JPEGEncode;
        tif->tif_encodestrip = JPEGEncode;
        tif->tif_encodetile = JPEGEncode;
        tif->tif_cleanup = JPEGCleanup;
        sp->defsparent = tif->tif_defstripsize;
        tif->tif_defstripsize = JPEGDefaultStripSize;
        sp->deftparent = tif->tif_deftilesize;
        tif->tif_deftilesize = JPEGDefaultTileSize;
        tif->tif_flags |= TIFF_NOBITREV;        /* no bit reversal, please */

        /*
         * Initialize libjpeg.
         */
        if (tif->tif_mode == O_RDONLY) {
                if (!TIFFjpeg_create_decompress(sp))
                        return (0);
        } else {
                if (!TIFFjpeg_create_compress(sp))
                        return (0);
        }

        return (1);
}
```

---

kfax'LZWSetupDecode() (./kdegraphics/kfax/libtiffax/tif_lzw.c:191)

```
LZWSetupDecode(TIFF* tif)
{
        LZWDecodeState* sp = DecoderState(tif);
        static char module[] = " LZWSetupDecode";
        int code;

        assert(sp != NULL);
        if (sp->dec_codetab == NULL) {
                sp->dec_codetab = (code_t*)_TIFFmalloc(CSIZE*sizeof (code_t));
                if (sp->dec_codetab == NULL) {
                        TIFFError(module, "No space for LZW code table");
                        return (0);
                }
                /*
                 * Pre-load the table.
                 */
                for (code = 255; code >= 0; code--) {
                        sp->dec_codetab[code].value = code;
                        sp->dec_codetab[code].firstchar = code;
                        sp->dec_codetab[code].length = 1;
                        sp->dec_codetab[code].next = NULL;
                }
        }
        return (1);
}

/*
 * Setup state for decoding a strip.
 */
```

```
static int
```

---

kfax'LZWPreDecode() (./kdegraphics/kfax/libtiffax/tif_lzw.c:221)

```
LZWPreDecode(TIFF* tif, tsample_t s)
{
        LZWDecodeState *sp = DecoderState(tif);

        (void) s;
        assert(sp != NULL);
        /*
         * Check for old bit-reversed codes.
         */
        if (tif->tif_rawdata[0] == 0 && (tif->tif_rawdata[1] & 0x1)) {
#ifdef LZW_COMPAT
                if (!sp->dec_decode) {
                        TIFFWarning(tif->tif_name,
                            "Old-style LZW codes, convert file");
                        /*
                         * Override default decoding methods with
                         * ones that deal with the old coding.
                         * Otherwise the predictor versions set
                         * above will call the compatibility routines
                         * through the dec_decode method.
                         */
                        tif->tif_decoderow = LZWDecodeCompat;
                        tif->tif_decodestrip = LZWDecodeCompat;
                        tif->tif_decodetile = LZWDecodeCompat;
                        /*
                         * If doing horizontal differencing, must
                         * re-setup the predictor logic since we
                         * switched the basic decoder methods...
                         */
                        (*tif->tif_setupdecode)(tif);
                        sp->dec_decode = LZWDecodeCompat;
                }
                sp->lzw_maxcode = MAXCODE(BITS_MIN);
#else /* !LZW_COMPAT */
                if (!sp->dec_decode) {
                        TIFFError(tif->tif_name,
                            "Old-style LZW codes not supported");
                        sp->dec_decode = LZWDecode;
                }
                return (0);
#endif/* !LZW_COMPAT */
        } else {
                sp->lzw_maxcode = MAXCODE(BITS_MIN)-1;
                sp->dec_decode = LZWDecode;
        }
        sp->lzw_nbits = BITS_MIN;
        sp->lzw_nextbits = 0;
        sp->lzw_nextdata = 0;

        sp->dec_restart = 0;
        sp->dec_nbitsmask = MAXCODE(BITS_MIN);
#ifdef LZW_CHECKEOS
        sp->dec_bitsleft = tif->tif_rawcc << 3;
#endif
```

```
        sp->dec_free_entp = sp->dec_codetab + CODE_FIRST;
        /*
         * Zero entries that are not yet filled in.  We do
         * this to guard against bogus input data that causes
         * us to index into undefined entries.  If you can
         * come up with a way to safely bounds-check input codes
         * while decoding then you can remove this operation.
         */
        _TIFFmemset(sp->dec_free_entp, 0, (CSIZE-CODE_FIRST)*sizeof (code_t));
        sp->dec_oldcodep = &sp->dec_codetab[-1];
        sp->dec_maxcodep = &sp->dec_codetab[sp->dec_nbitsmask-1];
        return (1);
}

/*
 * Decode a "hunk of data".
 */
```

---

kfax'codeLoop() (./kdegraphics/kfax/libtiffax/tif_lzw.c:304)

```
codeLoop(TIFF* tif)
{
        TIFFError(tif->tif_name,
            "LZWDecode: Bogus encoding, loop in the code table; scanline %d",
            tif->tif_row);
}

static int
```

---

kfax'LZWDecode() (./kdegraphics/kfax/libtiffax/tif_lzw.c:312)

```
LZWDecode(TIFF* tif, tidata_t op0, tsize_t occ0, tsample_t s)
{
        LZWDecodeState *sp = DecoderState(tif);
        char *op = (char*) op0;
        long occ = (long) occ0;
        char *tp;
        u_char *bp;
        hcode_t code;
        int len;
        long nbits, nextbits, nextdata, nbitsmask;
        code_t *codep, *free_entp, *maxcodep, *oldcodep;

        (void) s;
        assert(sp != NULL);
        /*
         * Restart interrupted output operation.
         */
        if (sp->dec_restart) {
                long residue;

                codep = sp->dec_codep;
                residue = codep->length - sp->dec_restart;
                if (residue > occ) {
                        /*
                         * Residue from previous decode is sufficient
```

```
	 * to satisfy decode request.  Skip to the
	 * start of the decoded string, place decoded
	 * values in the output buffer, and return.
	 */
	sp->dec_restart += occ;
	do {
		codep = codep->next;
	} while (--residue > occ && codep);
	if (codep) {
		tp = op + occ;
		do {
			*--tp = codep->value;
			codep = codep->next;
		} while (--occ && codep);
	}
	return (1);
}
/*
 * Residue satisfies only part of the decode request.
 */
op += residue, occ -= residue;
tp = op;
do {
	int t;
	--tp;
	t = codep->value;
	codep = codep->next;
	*tp = t;
} while (--residue && codep);
sp->dec_restart = 0;
}

bp = (u_char *)tif->tif_rawcp;
nbits = sp->lzw_nbits;
nextdata = sp->lzw_nextdata;
nextbits = sp->lzw_nextbits;
nbitsmask = sp->dec_nbitsmask;
oldcodep = sp->dec_oldcodep;
free_entp = sp->dec_free_entp;
maxcodep = sp->dec_maxcodep;

while (occ > 0) {
	NextCode(tif, sp, bp, code, GetNextCode);
	if (code == CODE_EOI)
		break;
	if (code == CODE_CLEAR) {
		free_entp = sp->dec_codetab + CODE_FIRST;
		nbits = BITS_MIN;
		nbitsmask = MAXCODE(BITS_MIN);
		maxcodep = sp->dec_codetab + nbitsmask-1;
		NextCode(tif, sp, bp, code, GetNextCode);
		if (code == CODE_EOI)
			break;
		*op++ = code, occ--;
		oldcodep = sp->dec_codetab + code;
		continue;
	}
	codep = sp->dec_codetab + code;

	/*
	 * Add the new entry to the code table.
```

```
		 */
		assert(&sp->dec_codetab[0] <= free_entp && free_entp < &sp->dec_
		free_entp->next = oldcodep;
		free_entp->firstchar = free_entp->next->firstchar;
		free_entp->length = free_entp->next->length+1;
		free_entp->value = (codep < free_entp) ?
		    codep->firstchar : free_entp->firstchar;
		if (++free_entp > maxcodep) {
			if (++nbits > BITS_MAX)		/* should not happen */
				nbits = BITS_MAX;
			nbitsmask = MAXCODE(nbits);
			maxcodep = sp->dec_codetab + nbitsmask-1;
		}
		oldcodep = codep;
		if (code >= 256) {
			/*
			 * Code maps to a string, copy string
			 * value to output (written in reverse).
			 */
			if (codep->length > occ) {
				/*
				 * String is too long for decode buffer,
				 * locate portion that will fit, copy to
				 * the decode buffer, and setup restart
				 * logic for the next decoding call.
				 */
				sp->dec_codep = codep;
				do {
					codep = codep->next;
				} while (codep && codep->length > occ);
				if (codep) {
					sp->dec_restart = occ;
					tp = op + occ;
					do  {
						*--tp = codep->value;
						codep = codep->next;
					}  while (--occ && codep);
					if (codep)
						codeLoop(tif);
				}
				break;
			}
			len = codep->length;
			tp = op + len;
			do {
				int t;
				--tp;
				t = codep->value;
				codep = codep->next;
				*tp = t;
			} while (codep && tp > op);
			if (codep) {
			    codeLoop(tif);
			    break;
			}
			op += len, occ -= len;
		} else
			*op++ = code, occ--;
	}

	tif->tif_rawcp = (tidata_t) bp;
```

```
        sp->lzw_nbits = (u_short) nbits;
        sp->lzw_nextdata = nextdata;
        sp->lzw_nextbits = nextbits;
        sp->dec_nbitsmask = nbitsmask;
        sp->dec_oldcodep = oldcodep;
        sp->dec_free_entp = free_entp;
        sp->dec_maxcodep = maxcodep;

        if (occ > 0) {
                TIFFError(tif->tif_name,
                "LZWDecode: Not enough data at scanline %d (short %d bytes)",
                    tif->tif_row, occ);
                return (0);
        }
        return (1);
}

#ifdef LZW_COMPAT
/*
 * Decode a "hunk of data" for old images.
 */
```

---

kfax'LZWDecodeCompat() (./kdegraphics/kfax/libtiffax/tif_lzw.c:493)

```
LZWDecodeCompat(TIFF* tif, tidata_t op0, tsize_t occ0, tsample_t s)
{
        LZWDecodeState *sp = DecoderState(tif);
        char *op = (char*) op0;
        long occ = (long) occ0;
        char *tp;
        u_char *bp;
        int code, nbits;
        long nextbits, nextdata, nbitsmask;
        code_t *codep, *free_entp, *maxcodep, *oldcodep;

        (void) s;
        assert(sp != NULL);
        /*
         * Restart interrupted output operation.
         */
        if (sp->dec_restart) {
                long residue;

                codep = sp->dec_codep;
                residue = codep->length - sp->dec_restart;
                if (residue > occ) {
                        /*
                         * Residue from previous decode is sufficient
                         * to satisfy decode request.  Skip to the
                         * start of the decoded string, place decoded
                         * values in the output buffer, and return.
                         */
                        sp->dec_restart += occ;
                        do {
                                codep = codep->next;
                        } while (--residue > occ);
                        tp = op + occ;
                        do {
```

```
                                *--tp = codep->value;
                                codep = codep->next;
                        } while (--occ);
                        return (1);
                }
                /*
                 * Residue satisfies only part of the decode request.
                 */
                op += residue, occ -= residue;
                tp = op;
                do {
                        *--tp = codep->value;
                        codep = codep->next;
                } while (--residue);
                sp->dec_restart = 0;
        }

        bp = (u_char *)tif->tif_rawcp;
        nbits = sp->lzw_nbits;
        nextdata = sp->lzw_nextdata;
        nextbits = sp->lzw_nextbits;
        nbitsmask = sp->dec_nbitsmask;
        oldcodep = sp->dec_oldcodep;
        free_entp = sp->dec_free_entp;
        maxcodep = sp->dec_maxcodep;

        while (occ > 0) {
                NextCode(tif, sp, bp, code, GetNextCodeCompat);
                if (code == CODE_EOI)
                        break;
                if (code == CODE_CLEAR) {
                        free_entp = sp->dec_codetab + CODE_FIRST;
                        nbits = BITS_MIN;
                        nbitsmask = MAXCODE(BITS_MIN);
                        maxcodep = sp->dec_codetab + nbitsmask;
                        NextCode(tif, sp, bp, code, GetNextCodeCompat);
                        if (code == CODE_EOI)
                                break;
                        *op++ = code, occ--;
                        oldcodep = sp->dec_codetab + code;
                        continue;
                }
                codep = sp->dec_codetab + code;

                /*
                 * Add the new entry to the code table.
                 */
                assert(&sp->dec_codetab[0] <= free_entp && free_entp < &sp->dec_
                free_entp->next = oldcodep;
                free_entp->firstchar = free_entp->next->firstchar;
                free_entp->length = free_entp->next->length+1;
                free_entp->value = (codep < free_entp) ?
                    codep->firstchar : free_entp->firstchar;
                if (++free_entp > maxcodep) {
                        if (++nbits > BITS_MAX)         /* should not happen */
                                nbits = BITS_MAX;
                        nbitsmask = MAXCODE(nbits);
                        maxcodep = sp->dec_codetab + nbitsmask;
                }
                oldcodep = codep;
                if (code >= 256) {
```

```
                        /*
                         * Code maps to a string, copy string
                         * value to output (written in reverse).
                         */
                        if (codep->length > occ) {
                                /*
                                 * String is too long for decode buffer,
                                 * locate portion that will fit, copy to
                                 * the decode buffer, and setup restart
                                 * logic for the next decoding call.
                                 */
                                sp->dec_codep = codep;
                                do {
                                        codep = codep->next;
                                } while (codep->length > occ);
                                sp->dec_restart = occ;
                                tp = op + occ;
                                do  {
                                        *--tp = codep->value;
                                        codep = codep->next;
                                }  while (--occ);
                                break;
                        }
                        op += codep->length, occ -= codep->length;
                        tp = op;
                        do {
                                *--tp = codep->value;
                        } while (codep = codep->next);
                } else
                        *op++ = code, occ--;
        }

        tif->tif_rawcp = (tidata_t) bp;
        sp->lzw_nbits = nbits;
        sp->lzw_nextdata = nextdata;
        sp->lzw_nextbits = nextbits;
        sp->dec_nbitsmask = nbitsmask;
        sp->dec_oldcodep = oldcodep;
        sp->dec_free_entp = free_entp;
        sp->dec_maxcodep = maxcodep;

        if (occ > 0) {
                TIFFError(tif->tif_name,
            "LZWDecodeCompat: Not enough data at scanline %d (short %d bytes)",
                    tif->tif_row, occ);
                return (0);
        }
        return (1);
}
#endif /* LZW_COMPAT */

/*
 * LZW Encoding.
 */

static int
```

---

kfax'LZWSetupEncode() (./kdegraphics/kfax/libtiffax/tif_lzw.c:644)

```
LZWSetupEncode(TIFF* tif)
{
        LZWEncodeState* sp = EncoderState(tif);
        static char module[] = "LZWSetupEncode";

        assert(sp != NULL);
        sp->enc_hashtab = (hash_t*) _TIFFmalloc(HSIZE*sizeof (hash_t));
        if (sp->enc_hashtab == NULL) {
                TIFFError(module, "No space for LZW hash table");
                return (0);
        }
        return (1);
}

/*
 * Reset encoding state at the start of a strip.
 */
static int
```

---

kfax'LZWPreEncode() (./kdegraphics/kfax/libtiffax/tif_lzw.c:662)

```
LZWPreEncode(TIFF* tif, tsample_t s)
{
        LZWEncodeState *sp = EncoderState(tif);

        (void) s;
        assert(sp != NULL);
        sp->lzw_nbits = BITS_MIN;
        sp->lzw_maxcode = MAXCODE(BITS_MIN);
        sp->lzw_free_ent = CODE_FIRST;
        sp->lzw_nextbits = 0;
        sp->lzw_nextdata = 0;
        sp->enc_checkpoint = CHECK_GAP;
        sp->enc_ratio = 0;
        sp->enc_incount = 0;
        sp->enc_outcount = 0;
        /*
         * The 4 here insures there is space for 2 max-sized
         * codes in LZWEncode and LZWPostDecode.
         */
        sp->enc_rawlimit = tif->tif_rawdata + tif->tif_rawdatasize-1 - 4;
        cl_hash(sp);            /* clear hash table */
        sp->enc_oldcode = (hcode_t) -1; /* generates CODE_CLEAR in LZWEncode */
        return (1);
}
```

---

kfax'LZWEncode() (./kdegraphics/kfax/libtiffax/tif_lzw.c:721)

```
LZWEncode(TIFF* tif, tidata_t bp, tsize_t cc, tsample_t s)
{
        register LZWEncodeState *sp = EncoderState(tif);
        register long fcode;
        register hash_t *hp;
        register int h, c;
```

```
	hcode_t ent;
	long disp;
	long incount, outcount, checkpoint;
	long nextdata, nextbits;
	int free_ent, maxcode, nbits;
	tidata_t op, limit;

	(void) s;
	if (sp == NULL)
		return (0);
	/*
	 * Load local state.
	 */
	incount = sp->enc_incount;
	outcount = sp->enc_outcount;
	checkpoint = sp->enc_checkpoint;
	nextdata = sp->lzw_nextdata;
	nextbits = sp->lzw_nextbits;
	free_ent = sp->lzw_free_ent;
	maxcode = sp->lzw_maxcode;
	nbits = sp->lzw_nbits;
	op = tif->tif_rawcp;
	limit = sp->enc_rawlimit;
	ent = sp->enc_oldcode;

	if (ent == (hcode_t) -1 && cc > 0) {
		/*
		 * NB: This is safe because it can only happen
		 *     at the start of a strip where we know there
		 *     is space in the data buffer.
		 */
		PutNextCode(op, CODE_CLEAR);
		ent = *bp++; cc--; incount++;
	}
	while (cc > 0) {
		c = *bp++; cc--; incount++;
		fcode = ((long)c << BITS_MAX) + ent;
		h = (c << HSHIFT) ^ ent;	/* xor hashing */
#ifdef _WINDOWS
		/*
		 * Check hash index for an overflow.
		 */
		if (h >= HSIZE)
			h -= HSIZE;
#endif
		hp = &sp->enc_hashtab[h];
		if (hp->hash == fcode) {
			ent = hp->code;
			continue;
		}
		if (hp->hash >= 0) {
			/*
			 * Primary hash failed, check secondary hash.
			 */
			disp = HSIZE - h;
			if (h == 0)
				disp = 1;
			do {
#ifndef _WINDOWS
				if ((hp -= disp) < sp->enc_hashtab)
					hp += HSIZE;
```

```
#else
                                /*
                                 * Avoid pointer arithmetic 'cuz of
                                 * wraparound problems with segments.
                                 */
                                if ((h -= disp) < 0)
                                        h += HSIZE;
                                hp = &sp->enc_hashtab[h];
#endif
                                if (hp->hash == fcode) {
                                        ent = hp->code;
                                        goto hit;
                                }
                        } while (hp->hash >= 0);
                }
                /*
                 * New entry, emit code and add to table.
                 */
                /*
                 * Verify there is space in the buffer for the code
                 * and any potential Clear code that might be emitted
                 * below.  The value of limit is setup so that there
                 * are at least 4 bytes free--room for 2 codes.
                 */
                if (op > limit) {
                        tif->tif_rawcc = (tsize_t)(op - tif->tif_rawdata);
                        TIFFFlushData1(tif);
                        op = tif->tif_rawdata;
                }
                PutNextCode(op, ent);
                ent = c;
                hp->code = free_ent++;
                hp->hash = fcode;
                if (free_ent == CODE_MAX-1) {
                        /* table is full, emit clear code and reset */
                        cl_hash(sp);
                        sp->enc_ratio = 0;
                        incount = 0;
                        outcount = 0;
                        free_ent = CODE_FIRST;
                        PutNextCode(op, CODE_CLEAR);
                        nbits = BITS_MIN;
                        maxcode = MAXCODE(BITS_MIN);
                } else {
                        /*
                         * If the next entry is going to be too big for
                         * the code size, then increase it, if possible.
                         */
                        if (free_ent > maxcode) {
                                nbits++;
                                assert(nbits <= BITS_MAX);
                                maxcode = (int) MAXCODE(nbits);
                        } else if (incount >= checkpoint) {
                                long rat;
                                /*
                                 * Check compression ratio and, if things seem
                                 * to be slipping, clear the hash table and
                                 * reset state.  The compression ratio is a
                                 * 24+8-bit fractional number.
                                 */
                                checkpoint = incount+CHECK_GAP;
```

```
                                CALCRATIO(sp, rat);
                                if (rat <= sp->enc_ratio) {
                                        cl_hash(sp);
                                        sp->enc_ratio = 0;
                                        incount = 0;
                                        outcount = 0;
                                        free_ent = CODE_FIRST;
                                        PutNextCode(op, CODE_CLEAR);
                                        nbits = BITS_MIN;
                                        maxcode = MAXCODE(BITS_MIN);
                                } else
                                        sp->enc_ratio = rat;
                        }
                }
        hit:
                ;
        }

        /*
         * Restore global state.
         */
        sp->enc_incount = incount;
        sp->enc_outcount = outcount;
        sp->enc_checkpoint = checkpoint;
        sp->enc_oldcode = ent;
        sp->lzw_nextdata = nextdata;
        sp->lzw_nextbits = nextbits;
        sp->lzw_free_ent = free_ent;
        sp->lzw_maxcode = maxcode;
        sp->lzw_nbits = nbits;
        tif->tif_rawcp = op;
        return (1);
}

/*
 * Finish off an encoded strip by flushing the last
 * string and tacking on an End Of Information code.
 */
static int
```

---

kfax'LZWPostEncode() (./kdegraphics/kfax/libtiffax/tif_lzw.c:888)

```
LZWPostEncode(TIFF* tif)
{
        register LZWEncodeState *sp = EncoderState(tif);
        tidata_t op = tif->tif_rawcp;
        long nextbits = sp->lzw_nextbits;
        long nextdata = sp->lzw_nextdata;
        long outcount = sp->enc_outcount;
        int nbits = sp->lzw_nbits;

        if (op > sp->enc_rawlimit) {
                tif->tif_rawcc = (tsize_t)(op - tif->tif_rawdata);
                TIFFFlushData1(tif);
                op = tif->tif_rawdata;
        }
        if (sp->enc_oldcode != (hcode_t) -1) {
                PutNextCode(op, sp->enc_oldcode);
```

```
                sp->enc_oldcode = (hcode_t) -1;
        }
        PutNextCode(op, CODE_EOI);
        if (nextbits > 0)
                *op++ = (u_char)(nextdata << (8-nextbits));
        tif->tif_rawcc = (tsize_t)(op - tif->tif_rawdata);
        return (1);
}

/*
 * Reset encoding hash table.
 */
static void
```

---

kfax'cl_hash() (./kdegraphics/kfax/libtiffax/tif_lzw.c:917)

```
cl_hash(LZWEncodeState* sp)
{
        register hash_t *hp = &sp->enc_hashtab[HSIZE-1];
        register long i = HSIZE-8;

        do {
                i -= 8;
                hp[-7].hash = -1;
                hp[-6].hash = -1;
                hp[-5].hash = -1;
                hp[-4].hash = -1;
                hp[-3].hash = -1;
                hp[-2].hash = -1;
                hp[-1].hash = -1;
                hp[ 0].hash = -1;
                hp -= 8;
        } while (i >= 0);
        for (i += 8; i > 0; i--, hp--)
                hp->hash = -1;
}

static void
```

---

kfax'LZWCleanup() (./kdegraphics/kfax/libtiffax/tif_lzw.c:939)

```
LZWCleanup(TIFF* tif)
{
        if (tif->tif_data) {
                if (tif->tif_mode == O_RDONLY) {
                        if (DecoderState(tif)->dec_codetab)
                                _TIFFfree(DecoderState(tif)->dec_codetab);
                } else {
                        if (EncoderState(tif)->enc_hashtab)
                                _TIFFfree(EncoderState(tif)->enc_hashtab);
                }
                _TIFFfree(tif->tif_data);
                tif->tif_data = NULL;
        }
}
```

```
int
```

kfax'TIFFInitLZW() (./kdegraphics/kfax/libtiffax/tif_lzw.c:955)

```
TIFFInitLZW(TIFF* tif, int scheme)
{
        assert(scheme == COMPRESSION_LZW);
        /*
         * Allocate state block so tag methods have storage to record values.
         */
        if (tif->tif_mode == O_RDONLY) {
                tif->tif_data = (tidata_t) _TIFFmalloc(sizeof (LZWDecodeState));
                if (tif->tif_data == NULL)
                        goto bad;
                DecoderState(tif)->dec_codetab = NULL;
                DecoderState(tif)->dec_decode = NULL;
        } else {
                tif->tif_data = (tidata_t) _TIFFmalloc(sizeof (LZWEncodeState));
                if (tif->tif_data == NULL)
                        goto bad;
                EncoderState(tif)->enc_hashtab = NULL;
        }
        /*
         * Install codec methods.
         */
        tif->tif_setupdecode = LZWSetupDecode;
        tif->tif_predecode = LZWPreDecode;
        tif->tif_decoderow = LZWDecode;
        tif->tif_decodestrip = LZWDecode;
        tif->tif_decodetile = LZWDecode;
        tif->tif_setupencode = LZWSetupEncode;
        tif->tif_preencode = LZWPreEncode;
        tif->tif_postencode = LZWPostEncode;
        tif->tif_encoderow = LZWEncode;
        tif->tif_encodestrip = LZWEncode;
        tif->tif_encodetile = LZWEncode;
        tif->tif_cleanup = LZWCleanup;
        /*
         * Setup predictor setup.
         */
        (void) TIFFPredictorInit(tif);
        return (1);
bad:
        TIFFError("TIFFInitLZW", "No space for LZW state block");
        return (0);
}

/*
 * Copyright (c) 1985, 1986 The Regents of the University of California.
 * All rights reserved.
 *
 * This code is derived from software contributed to Berkeley by
 * James A. Woods, derived from original work by Spencer Thomas
 * and Joseph Orost.
 *
 * Redistribution and use in source and binary forms are permitted
 * provided that the above copyright notice and this paragraph are
 * duplicated in all such forms and that any documentation,
```

```
 * advertising materials, and other materials related to such
 * distribution and use acknowledge that the software was developed
 * by the University of California, Berkeley.  The name of the
 * University may not be used to endorse or promote products derived
 * from this software without specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
 * WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */
```

---

kfax'_tiffReadProc() (./kdegraphics/kfax/libtiffax/tif_msdos.c:37)

```
_tiffReadProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        return (read((int) fd, buf, size));
}

static tsize_t
```

---

kfax'_tiffWriteProc() (./kdegraphics/kfax/libtiffax/tif_msdos.c:43)

```
_tiffWriteProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        return (write((int) fd, buf, size));
}

static toff_t
```

---

kfax'_tiffSeekProc() (./kdegraphics/kfax/libtiffax/tif_msdos.c:49)

```
_tiffSeekProc(thandle_t fd, toff_t off, int whence)
{
        return (lseek((int) fd, (off_t) off, whence));
}

static int
```

---

kfax'_tiffCloseProc() (./kdegraphics/kfax/libtiffax/tif_msdos.c:55)

```
_tiffCloseProc(thandle_t fd)
{
        return (close((int) fd));
}

#include <sys/stat.h>

static toff_t
```

---

kfax'_tiffSizeProc() (./kdegraphics/kfax/libtiffax/tif_msdos.c:63)

```
_tiffSizeProc(thandle_t fd)
{
        struct stat sb;
        return (fstat((int) fd, &sb) < 0 ? 0 : sb.st_size);
}

static int
```

---

kfax'_tiffMapProc() (./kdegraphics/kfax/libtiffax/tif_msdos.c:70)

```
_tiffMapProc(thandle_t fd, tdata_t* pbase, toff_t* psize)
{
        return (0);
}

static void
```

---

kfax'_tiffUnmapProc() (./kdegraphics/kfax/libtiffax/tif_msdos.c:76)

```
_tiffUnmapProc(thandle_t fd, tdata_t base, toff_t size)
{
}

/*
 * Open a TIFF file descriptor for read/writing.
 */
TIFF*
```

---

kfax'TIFFFdOpen() (./kdegraphics/kfax/libtiffax/tif_msdos.c:84)

```
TIFFFdOpen(int fd, const char* name, const char* mode)
{
        TIFF* tif;

        tif = TIFFClientOpen(name, mode,
            (void*) fd,
            _tiffReadProc, _tiffWriteProc, _tiffSeekProc, _tiffCloseProc,
            _tiffSizeProc, _tiffMapProc, _tiffUnmapProc);
        if (tif)
                tif->tif_fd = fd;
        return (tif);
}

/*
 * Open a TIFF file for read/writing.
 */
TIFF*
```

---

kfax'TIFFOpen() (./kdegraphics/kfax/libtiffax/tif_msdos.c:101)

```
TIFFOpen(const char* name, const char* mode)
{
        static const char module[] = "TIFFOpen";
        int m, fd;

        m = _TIFFgetMode(mode, module);
        if (m == -1)
                return ((TIFF*)0);
        fd = open(name, m|O_BINARY, 0666);
        if (fd < 0) {
                TIFFError(module, "%s: Cannot open", name);
                return ((TIFF*)0);
        }
        return (TIFFFdOpen(fd, name, mode));
}
```

---

kfax'_TIFFmalloc() (./kdegraphics/kfax/libtiffax/tif_msdos.c:125)

```
_TIFFmalloc(tsize_t s)
{
        return (malloc((size_t) s));
}

void
```

---

kfax'_TIFFfree() (./kdegraphics/kfax/libtiffax/tif_msdos.c:131)

```
_TIFFfree(tdata_t p)
{
        free(p);
}

tdata_t
```

---

kfax'_TIFFrealloc() (./kdegraphics/kfax/libtiffax/tif_msdos.c:137)

```
_TIFFrealloc(tdata_t p, tsize_t s)
{
        return (realloc(p, (size_t) s));
}

void
```

---

kfax'_TIFFmemset() (./kdegraphics/kfax/libtiffax/tif_msdos.c:143)

```
_TIFFmemset(tdata_t p, int v, tsize_t c)
{
```

```
        memset(p, v, (size_t) c);
}

void
```

---

kfax'_TIFFmemcpy() (./kdegraphics/kfax/libtiffax/tif_msdos.c:149)

```
_TIFFmemcpy(tdata_t d, const tdata_t s, tsize_t c)
{
        memcpy(d, s, (size_t) c);
}

int
```

---

kfax'_TIFFmemcmp() (./kdegraphics/kfax/libtiffax/tif_msdos.c:155)

```
_TIFFmemcmp(const tdata_t p1, const tdata_t p2, tsize_t c)
{
        return (memcmp(p1, p2, (size_t) c));
}

static void
```

---

kfax'msdosWarningHandler() (./kdegraphics/kfax/libtiffax/tif_msdos.c:161)

```
msdosWarningHandler(const char* module, const char* fmt, va_list ap)
{
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        fprintf(stderr, "Warning, ");
        vfprintf(stderr, fmt, ap);
        fprintf(stderr, ".\n");
}
```

---

kfax'msdosErrorHandler() (./kdegraphics/kfax/libtiffax/tif_msdos.c:172)

```
msdosErrorHandler(const char* module, const char* fmt, va_list ap)
{
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        vfprintf(stderr, fmt, ap);
        fprintf(stderr, ".\n");
}
```

---

kfax'NeXTDecode() (./kdegraphics/kfax/libtiffax/tif_next.c:49)

```
NeXTDecode(TIFF* tif, tidata_t buf, tsize_t occ, tsample_t s)
{
```

```
	register u_char *bp, *op;
	register tsize_t cc;
	register int n;
	tidata_t row;
	tsize_t scanline;

	(void) s;
	/*
	 * Each scanline is assumed to start off as all
	 * white (we assume a PhotometricInterpretation
	 * of ``min-is-black'').
	 */
	for (op = buf, cc = occ; cc-- > 0;)
		*op++ = 0xff;

	bp = (u_char *)tif->tif_rawcp;
	cc = tif->tif_rawcc;
	scanline = tif->tif_scanlinesize;
	for (row = buf; (long)occ > 0; occ -= scanline, row += scanline) {
		n = *bp++, cc--;
		switch (n) {
		case LITERALROW:
			/*
			 * The entire scanline is given as literal values.
			 */
			if (cc < scanline)
				goto bad;
			_TIFFmemcpy(row, bp, scanline);
			bp += scanline;
			cc -= scanline;
			break;
		case LITERALSPAN: {
			int off;
			/*
			 * The scanline has a literal span
			 * that begins at some offset.
			 */
			off = (bp[0] * 256) + bp[1];
			n = (bp[2] * 256) + bp[3];
			if (cc < 4+n)
				goto bad;
			_TIFFmemcpy(row+off, bp+4, n);
			bp += 4+n;
			cc -= 4+n;
			break;
		}
		default: {
			register int npixels = 0, grey;
			u_long imagewidth = tif->tif_dir.td_imagewidth;

			/*
			 * The scanline is composed of a sequence
			 * of constant color ``runs''.  We shift
			 * into ``run mode'' and interpret bytes
			 * as codes of the form <color><npixels>
			 * until we've filled the scanline.
			 */
			op = row;
			for (;;) {
				grey = (n>>6) & 0x3;
				n &= 0x3f;
```

```
                                while (n-- > 0)
                                        SETPIXEL(op, grey);
                                if (npixels >= imagewidth)
                                        break;
                                if (cc == 0)
                                        goto bad;
                                n = *bp++, cc--;
                        }
                        break;
                }
                }
        }
        tif->tif_rawcp = (tidata_t) bp;
        tif->tif_rawcc = cc;
        return (1);
bad:
        TIFFError(tif->tif_name, "NeXTDecode: Not enough data for scanline %ld",
            (long) tif->tif_row);
        return (0);
}

int
```

kfax'TIFFInitNeXT() (./kdegraphics/kfax/libtiffax/tif_next.c:134)

```
TIFFInitNeXT(TIFF* tif, int scheme)
{
        (void) scheme;
        tif->tif_decoderow = NeXTDecode;
        tif->tif_decodestrip = NeXTDecode;
        tif->tif_decodetile = NeXTDecode;
        return (1);
}
```

kfax'TIFFInitOrder() (./kdegraphics/kfax/libtiffax/tif_open.c:85)

```
TIFFInitOrder(register TIFF* tif, int magic, int bigendian)
{
#ifdef notdef
        /*
         * NB: too many applications assume that data is returned
         *     by the library in MSB2LSB bit order to change the
         *     default bit order to reflect the native cpu.  This
         *     may change in the future in which case applications
         *     will need to check the value of the FillOrder tag.
         */
        tif->tif_flags = (tif->tif_flags &~ TIFF_FILLORDER) | HOST_FILLORDER;
#else
        tif->tif_flags = (tif->tif_flags &~ TIFF_FILLORDER) | FILLORDER_MSB2LSB;
#endif

        tif->tif_typemask = typemask;
        if (magic == TIFF_BIGENDIAN) {
                tif->tif_typeshift = bigTypeshift;
                if (!bigendian)
                        tif->tif_flags |= TIFF_SWAB;
```

```
        } else {
                tif->tif_typeshift = litTypeshift;
                if (bigendian)
                        tif->tif_flags |= TIFF_SWAB;
        }
}

int
```

---

kfax'_TIFFgetMode() (./kdegraphics/kfax/libtiffax/tif_open.c:113)

```
_TIFFgetMode(const char* mode, const char* module)
{
        int m = -1;

        switch (mode[0]) {
        case 'r':
                m = O_RDONLY;
                if (mode[1] == '+')
                        m = O_RDWR;
                break;
        case 'w':
        case 'a':
                m = O_RDWR|O_CREAT;
                if (mode[0] == 'w')
                        m |= O_TRUNC;
                break;
        default:
                TIFFError(module, "\"%s\": Bad mode", mode);
                break;
        }
        return (m);
}

TIFF*
```

---

kfax'TIFFClientOpen() (./kdegraphics/kfax/libtiffax/tif_open.c:137)

```
TIFFClientOpen(
        const char* name, const char* mode,
        thandle_t clientdata,
        TIFFReadWriteProc readproc,
        TIFFReadWriteProc writeproc,
        TIFFSeekProc seekproc,
        TIFFCloseProc closeproc,
        TIFFSizeProc sizeproc,
        TIFFMapFileProc mapproc,
        TIFFUnmapFileProc unmapproc
)
{
        static const char module[] = "TIFFClientOpen";
        TIFF *tif;
        int m, bigendian;

        m = _TIFFgetMode(mode, module);
        if (m == -1)
```

```
                goto bad2;
        tif = (TIFF *)_TIFFmalloc(sizeof (TIFF) + strlen(name) + 1);
        if (tif == NULL) {
                TIFFError(module, "%s: Out of memory (TIFF structure)", name);
                goto bad2;
        }
        _TIFFmemset(tif, 0, sizeof (*tif));
        tif->tif_name = (char *)tif + sizeof (TIFF);
        strcpy(tif->tif_name, name);
        tif->tif_mode = m &~ (O_CREAT|O_TRUNC);
        tif->tif_curdir = (tdir_t) -1;          /* non-existent directory */
        tif->tif_curoff = 0;
        tif->tif_curstrip = (tstrip_t) -1;      /* invalid strip */
        tif->tif_row = (uint32)-1;              /* read/write pre-increment */
        tif->tif_clientdata = clientdata;
        tif->tif_readproc = readproc;
        tif->tif_writeproc = writeproc;
        tif->tif_seekproc = seekproc;
        tif->tif_closeproc = closeproc;
        tif->tif_sizeproc = sizeproc;
        tif->tif_mapproc = mapproc;
        tif->tif_unmapproc = unmapproc;

        { union { int32 i; char c[4]; } u; u.i = 1; bigendian = u.c[0] == 0; }
#ifdef ENDIANHACK_SUPPORT
        /*
         * Numerous vendors, typically on the PC, do not correctly
         * support TIFF; they only support the Intel little-endian
         * byte order.  If this hack is enabled, then applications
         * can open a file with a specific byte-order by specifying
         * either "wl" (for litt-endian byte order) or "wb" for
         * (big-endian byte order).  This support is not configured
         * by default because it supports the violation of the TIFF
         * spec that says that readers *MUST* support both byte orders.
         *
         * It is strongly recommended that you not use this feature
         * except to deal with busted apps that write invalid TIFF.
         * And even in those cases you should bang on the vendors to
         * fix their software.
         */
        if ((m&O_CREAT) &&
            ((bigendian && mode[1] == 'l') || (!bigendian && mode[1] == 'b')))
                tif->tif_flags |= TIFF_SWAB;
#endif
        /*
         * Read in TIFF header.
         */
        if (!ReadOK(tif, &tif->tif_header, sizeof (TIFFHeader))) {
                if (tif->tif_mode == O_RDONLY) {
                        TIFFError(name, "Cannot read TIFF header");
                        goto bad;
                }
                /*
                 * Setup header and write.
                 */
                tif->tif_header.tiff_magic = tif->tif_flags & TIFF_SWAB
                    ? (bigendian ? TIFF_LITTLEENDIAN : TIFF_BIGENDIAN)
                    : (bigendian ? TIFF_BIGENDIAN : TIFF_LITTLEENDIAN);
                tif->tif_header.tiff_version = TIFF_VERSION;
                tif->tif_header.tiff_diroff = 0;        /* filled in later */
                if (!WriteOK(tif, &tif->tif_header, sizeof (TIFFHeader))) {
```

```
                TIFFError(name, "Error writing TIFF header");
                goto bad;
        }
        /*
         * Setup the byte order handling.
         */
        TIFFInitOrder(tif, tif->tif_header.tiff_magic, bigendian);
        /*
         * Setup default directory.
         */
        if (!TIFFDefaultDirectory(tif))
                goto bad;
        tif->tif_diroff = 0;
        return (tif);
}
/*
 * Setup the byte order handling.
 */
if (tif->tif_header.tiff_magic != TIFF_BIGENDIAN &&
    tif->tif_header.tiff_magic != TIFF_LITTLEENDIAN) {
        TIFFError(name,  "Not a TIFF file, bad magic number %d (0x%x)",
            tif->tif_header.tiff_magic,
            tif->tif_header.tiff_magic);
        goto bad;
}
TIFFInitOrder(tif, tif->tif_header.tiff_magic, bigendian);
/*
 * Swap header if required.
 */
if (tif->tif_flags & TIFF_SWAB) {
        TIFFSwabShort(&tif->tif_header.tiff_version);
        TIFFSwabLong(&tif->tif_header.tiff_diroff);
}
/*
 * Now check version (if needed, it's been byte-swapped).
 * Note that this isn't actually a version number, it's a
 * magic number that doesn't change (stupid).
 */
if (tif->tif_header.tiff_version != TIFF_VERSION) {
        TIFFError(name,
            "Not a TIFF file, bad version number %d (0x%x)",
            tif->tif_header.tiff_version,
            tif->tif_header.tiff_version);
        goto bad;
}
tif->tif_flags |= TIFF_MYBUFFER;
tif->tif_rawcp = tif->tif_rawdata = 0;
tif->tif_rawdatasize = 0;
/*
 * Setup initial directory.
 */
switch (mode[0]) {
case 'r':
        tif->tif_nextdiroff = tif->tif_header.tiff_diroff;
        if (TIFFMapFileContents(tif, (tdata_t*) &tif->tif_base, &tif->ti
                tif->tif_flags |= TIFF_MAPPED;
        if (TIFFReadDirectory(tif)) {
                tif->tif_rawcc = -1;
                tif->tif_flags |= TIFF_BUFFERSETUP;
                return (tif);
        }
```

```
                break;
        case 'a':
                /*
                 * New directories are automatically append
                 * to the end of the directory chain when they
                 * are written out (see TIFFWriteDirectory).
                 */
                if (!TIFFDefaultDirectory(tif))
                        goto bad;
                return (tif);
        }
bad:
        tif->tif_mode = O_RDONLY;       /* XXX avoid flush */
        TIFFClose(tif);
        return ((TIFF*)0);
bad2:
        (void) (*closeproc)(clientdata);
        return ((TIFF*)0);
}

/*
 * Query functions to access private data.
 */

/*
 * Return open file's name.
 */
const char *
```

---

kfax'TIFFFileName() (./kdegraphics/kfax/libtiffax/tif_open.c:305)

```
TIFFFileName(TIFF* tif)
{
        return (tif->tif_name);
}

/*
 * Return open file's I/O descriptor.
 */
int
```

---

kfax'TIFFFileno() (./kdegraphics/kfax/libtiffax/tif_open.c:314)

```
TIFFFileno(TIFF* tif)
{
        return (tif->tif_fd);
}

/*
 * Return read/write mode.
 */
int
```

---

kfax'TIFFGetMode() (./kdegraphics/kfax/libtiffax/tif_open.c:323)

```
TIFFGetMode(TIFF* tif)
{
        return (tif->tif_mode);
}

/*
 * Return nonzero if file is organized in
 * tiles; zero if organized as strips.
 */
int
```

---

kfax'TIFFIsTiled() (./kdegraphics/kfax/libtiffax/tif_open.c:333)

```
TIFFIsTiled(TIFF* tif)
{
        return (isTiled(tif));
}

/*
 * Return current row being read/written.
 */
uint32
```

---

kfax'TIFFCurrentRow() (./kdegraphics/kfax/libtiffax/tif_open.c:342)

```
TIFFCurrentRow(TIFF* tif)
{
        return (tif->tif_row);
}

/*
 * Return index of the current directory.
 */
tdir_t
```

---

kfax'TIFFCurrentDirectory() (./kdegraphics/kfax/libtiffax/tif_open.c:351)

```
TIFFCurrentDirectory(TIFF* tif)
{
        return (tif->tif_curdir);
}

/*
 * Return current strip.
 */
tstrip_t
```

---

kfax'TIFFCurrentStrip() (./kdegraphics/kfax/libtiffax/tif_open.c:360)

```
TIFFCurrentStrip(TIFF* tif)
{
        return (tif->tif_curstrip);
}

/*
 * Return current tile.
 */
ttile_t
```

---

kfax'TIFFCurrentTile() (./kdegraphics/kfax/libtiffax/tif_open.c:369)

```
TIFFCurrentTile(TIFF* tif)
{
        return (tif->tif_curtile);
}

/*
 * Return nonzero if the file has byte-swapped data.
 */
int
```

---

kfax'TIFFIsByteSwapped() (./kdegraphics/kfax/libtiffax/tif_open.c:378)

```
TIFFIsByteSwapped(TIFF* tif)
{
        return ((tif->tif_flags & TIFF_SWAB) != 0);
}

/*
 * Return nonzero if the data is returned up-sampled.
 */
int
```

---

kfax'TIFFIsUpSampled() (./kdegraphics/kfax/libtiffax/tif_open.c:387)

```
TIFFIsUpSampled(TIFF* tif)
{
        return (isUpSampled(tif));
}

/*
 * Return nonzero if the data is returned in MSB-to-LSB bit order.
 */
int
```

---

kfax'TIFFIsMSB2LSB() (./kdegraphics/kfax/libtiffax/tif_open.c:396)

```
TIFFIsMSB2LSB(TIFF* tif)
{
        return (isFillOrder(tif, FILLORDER_MSB2LSB));
}
```

---

kfax'PackBitsPreEncode() (./kdegraphics/kfax/libtiffax/tif_packbits.c:38)

```
PackBitsPreEncode(TIFF* tif, tsample_t s)
{
        (void) s;
        /*
         * Calculate the scanline/tile-width size in bytes.
         */
        if (isTiled(tif))
                tif->tif_data = (tidata_t) TIFFTileRowSize(tif);
        else
                tif->tif_data = (tidata_t) TIFFScanlineSize(tif);
        return (1);
}

/*
 * NB: tidata is the type representing *(tidata_t);
 *     if tidata_t is made signed then this type must
 *     be adjusted accordingly.
 */
```

---

kfax'PackBitsEncode() (./kdegraphics/kfax/libtiffax/tif_packbits.c:62)

```
PackBitsEncode(TIFF* tif, tidata_t buf, tsize_t cc, tsample_t s)
{
        u_char* bp = (u_char*) buf;
        tidata_t op, ep, lastliteral;
        long n, slop;
        int b;
        enum { BASE, LITERAL, RUN, LITERAL_RUN } state;

        (void) s;
        op = tif->tif_rawcp;
        ep = tif->tif_rawdata + tif->tif_rawdatasize;
        state = BASE;
        lastliteral = 0;
        while (cc > 0) {
                /*
                 * Find the longest string of identical bytes.
                 */
                b = *bp++, cc--, n = 1;
                for (; cc > 0 && b == *bp; cc--, bp++)
                        n++;
        again:
                if (op + 2 >= ep) {             /* insure space for new data */
                        /*
                         * Be careful about writing the last
                         * literal.  Must write up to that point
                         * and then copy the remainder to the
                         * front of the buffer.
```

```
		 */
		if (state == LITERAL || state == LITERAL_RUN) {
			slop = op - lastliteral;
			tif->tif_rawcc += lastliteral - tif->tif_rawcp;
			if (!TIFFFlushData1(tif))
				return (-1);
			op = tif->tif_rawcp;
			while (slop-- > 0)
				*op++ = *lastliteral++;
			lastliteral = tif->tif_rawcp;
		} else {
			tif->tif_rawcc += op - tif->tif_rawcp;
			if (!TIFFFlushData1(tif))
				return (-1);
			op = tif->tif_rawcp;
		}
	}
	switch (state) {
	case BASE:		/* initial state, set run/literal */
		if (n > 1) {
			state = RUN;
			if (n > 128) {
				*op++ = (tidata) -127;
				*op++ = b;
				n -= 128;
				goto again;
			}
			*op++ = (tidataval_t)(-(n-1));
			*op++ = b;
		} else {
			lastliteral = op;
			*op++ = 0;
			*op++ = b;
			state = LITERAL;
		}
		break;
	case LITERAL:		/* last object was literal string */
		if (n > 1) {
			state = LITERAL_RUN;
			if (n > 128) {
				*op++ = (tidata) -127;
				*op++ = b;
				n -= 128;
				goto again;
			}
			*op++ = (tidataval_t)(-(n-1));	/* encode run */
			*op++ = b;
		} else {			/* extend literal */
			if (++(*lastliteral) == 127)
				state = BASE;
			*op++ = b;
		}
		break;
	case RUN:		/* last object was run */
		if (n > 1) {
			if (n > 128) {
				*op++ = (tidata) -127;
				*op++ = b;
				n -= 128;
				goto again;
			}
```

```
                                *op++ = (tidataval_t)(-(n-1));
                                *op++ = b;
                        } else {
                                lastliteral = op;
                                *op++ = 0;
                                *op++ = b;
                                state = LITERAL;
                        }
                        break;
                case LITERAL_RUN:       /* literal followed by a run */
                        /*
                         * Check to see if previous run should
                         * be converted to a literal, in which
                         * case we convert literal-run-literal
                         * to a single literal.
                         */
                        if (n == 1 && op[-2] == (tidata) -1 &&
                            *lastliteral < 126) {
                                state = (((*lastliteral) += 2) == 127 ?
                                    BASE : LITERAL);
                                op[-2] = op[-1];        /* replicate */
                        } else
                                state = RUN;
                        goto again;
                }
        }
        tif->tif_rawcc += op - tif->tif_rawcp;
        tif->tif_rawcp = op;
        return (1);
}

/*
 * Encode a rectangular chunk of pixels.  We break it up
 * into row-sized pieces to insure that encoded runs do
 * not span rows.  Otherwise, there can be problems with
 * the decoder if data is read, for example, by scanlines
 * when it was encoded by strips.
 */
static int
```

---

kfax'PackBitsEncodeChunk() (./kdegraphics/kfax/libtiffax/tif_packbits.c:189)

```
PackBitsEncodeChunk(TIFF* tif, tidata_t bp, tsize_t cc, tsample_t s)
{
        tsize_t rowsize = (tsize_t) tif->tif_data;

        assert(rowsize > 0);
        while ((long)cc > 0) {
                if (PackBitsEncode(tif, bp, rowsize, s) < 0)
                        return (-1);
                bp += rowsize;
                cc -= rowsize;
        }
        return (1);
}

static int
```

kfax'PackBitsDecode() (./kdegraphics/kfax/libtiffax/tif_packbits.c:204)

```
PackBitsDecode(TIFF* tif, tidata_t op, tsize_t occ, tsample_t s)
{
        char *bp;
        tsize_t cc;
        long n;
        int b;

        (void) s;
        bp = (char*) tif->tif_rawcp;
        cc = tif->tif_rawcc;
        while (cc > 0 && (long)occ > 0) {
                n = (long) *bp++, cc--;
                /*
                 * Watch out for compilers that
                 * don't sign extend chars...
                 */
                if (n >= 128)
                        n -= 256;
                if (n < 0) {            /* replicate next byte -n+1 times */
                        if (n == -128)  /* nop */
                                continue;
                        n = -n + 1;
                        occ -= n;
                        b = *bp++, cc--;
                        while (n-- > 0)
                                *op++ = b;
                } else {                /* copy next n+1 bytes literally */
                        _TIFFmemcpy(op, bp, ++n);
                        op += n; occ -= n;
                        bp += n; cc -= n;
                }
        }
        tif->tif_rawcp = (tidata_t) bp;
        tif->tif_rawcc = cc;
        if (occ > 0) {
                TIFFError(tif->tif_name,
                    "PackBitsDecode: Not enough data for scanline %ld",
                    (long) tif->tif_row);
                return (0);
        }
        /* check for buffer overruns? */
        return (1);
}

int
```

kfax'TIFFInitPackBits() (./kdegraphics/kfax/libtiffax/tif_packbits.c:249)

```
TIFFInitPackBits(TIFF* tif, int scheme)
{
        (void) scheme;
        tif->tif_decoderow = PackBitsDecode;
        tif->tif_decodestrip = PackBitsDecode;
```

```
        tif->tif_decodetile = PackBitsDecode;
        tif->tif_preencode = PackBitsPreEncode;
        tif->tif_encoderow = PackBitsEncode;
        tif->tif_encodestrip = PackBitsEncodeChunk;
        tif->tif_encodetile = PackBitsEncodeChunk;
        return (1);
}
```

---

kfax'PredictorSetup() (./kdegraphics/kfax/libtiffax/tif_predict.c:50)

```
PredictorSetup(TIFF* tif)
{
        TIFFPredictorState* sp = PredictorState(tif);
        TIFFDirectory* td = &tif->tif_dir;

        if (sp->predictor == 1)         /* no differencing */
                return (1);
        if (sp->predictor != 2) {
                TIFFError(tif->tif_name, "\"Predictor\" value %d not supported",
                    sp->predictor);
                return (0);
        }
        if (td->td_bitspersample != 8 && td->td_bitspersample != 16) {
                TIFFError(tif->tif_name,
    "Horizontal differencing \"Predictor\" not supported with %d-bit samples",
                    td->td_bitspersample);
                return (0);
        }
        sp->stride = (td->td_planarconfig == PLANARCONFIG_CONTIG ?
            td->td_samplesperpixel : 1);
        /*
         * Calculate the scanline/tile-width size in bytes.
         */
        if (isTiled(tif))
                sp->rowsize = TIFFTileRowSize(tif);
        else
                sp->rowsize = TIFFScanlineSize(tif);
        return (1);
}

static int
```

---

kfax'PredictorSetupDecode() (./kdegraphics/kfax/libtiffax/tif_predict.c:81)

```
PredictorSetupDecode(TIFF* tif)
{
        TIFFPredictorState* sp = PredictorState(tif);
        TIFFDirectory* td = &tif->tif_dir;

        if (!(*sp->setupdecode)(tif) || !PredictorSetup(tif))
                return (0);
        if (sp->predictor == 2) {
                switch (td->td_bitspersample) {
                case 8:  sp->pfunc = horAcc8; break;
                case 16: sp->pfunc = horAcc16; break;
                }
```

```
                /*
                 * Override default decoding method with
                 * one that does the predictor stuff.
                 */
                sp->coderow = tif->tif_decoderow;
                tif->tif_decoderow = PredictorDecodeRow;
                sp->codestrip = tif->tif_decodestrip;
                tif->tif_decodestrip = PredictorDecodeTile;
                sp->codetile = tif->tif_decodetile;
                tif->tif_decodetile = PredictorDecodeTile;
                /*
                 * If the data is horizontally differenced
                 * 16-bit data that requires byte-swapping,
                 * then it must be byte swapped before the
                 * accumulation step.  We do this with a
                 * special-purpose routine and override the
                 * normal post decoding logic that the library
                 * setup when the directory was read.
                 */
                if (tif->tif_flags&TIFF_SWAB) {
                        if (sp->pfunc == horAcc16) {
                                sp->pfunc = swabHorAcc16;
                                tif->tif_postdecode = _TIFFNoPostDecode;
                        } /* else handle 32-bit case... */
                }
        }
        return (1);
}

static int
```

---

kfax'PredictorSetupEncode() (./kdegraphics/kfax/libtiffax/tif_predict.c:123)

```
PredictorSetupEncode(TIFF* tif)
{
        TIFFPredictorState* sp = PredictorState(tif);
        TIFFDirectory* td = &tif->tif_dir;

        if (!(*sp->setupencode)(tif) || !PredictorSetup(tif))
                return (0);
        if (sp->predictor == 2) {
                switch (td->td_bitspersample) {
                case 8:  sp->pfunc = horDiff8; break;
                case 16: sp->pfunc = horDiff16; break;
                }
                /*
                 * Override default encoding method with
                 * one that does the predictor stuff.
                 */
                sp->coderow = tif->tif_encoderow;
                tif->tif_encoderow = PredictorEncodeRow;
                sp->codestrip = tif->tif_encodestrip;
                tif->tif_encodestrip = PredictorEncodeTile;
                sp->codetile = tif->tif_encodetile;
                tif->tif_encodetile = PredictorEncodeTile;
        }
        return (1);
}
```

kfax'horAcc8() (./kdegraphics/kfax/libtiffax/tif_predict.c:167)

```
horAcc8(TIFF* tif, tidata_t cp0, tsize_t cc)
{
        TIFFPredictorState* sp = PredictorState(tif);
        u_int stride = sp->stride;

        char* cp = (char*) cp0;
        if (cc > stride) {
                cc -= stride;
                /*
                 * Pipeline the most common cases.
                 */
                if (stride == 3)  {
                        u_int cr = cp[0];
                        u_int cg = cp[1];
                        u_int cb = cp[2];
                        do {
                                cc -= 3, cp += 3;
                                cp[0] = (cr += cp[0]);
                                cp[1] = (cg += cp[1]);
                                cp[2] = (cb += cp[2]);
                        } while ((int32) cc > 0);
                } else if (stride == 4)  {
                        u_int cr = cp[0];
                        u_int cg = cp[1];
                        u_int cb = cp[2];
                        u_int ca = cp[3];
                        do {
                                cc -= 4, cp += 4;
                                cp[0] = (cr += cp[0]);
                                cp[1] = (cg += cp[1]);
                                cp[2] = (cb += cp[2]);
                                cp[3] = (ca += cp[3]);
                        } while ((int32) cc > 0);
                } else  {
                        do {
                                XREPEAT4(stride, cp[stride] += *cp; cp++)
                                cc -= stride;
                        } while ((int32) cc > 0);
                }
        }
}

static void
```

kfax'swabHorAcc16() (./kdegraphics/kfax/libtiffax/tif_predict.c:210)

```
swabHorAcc16(TIFF* tif, tidata_t cp0, tsize_t cc)
{
        TIFFPredictorState* sp = PredictorState(tif);
        u_int stride = sp->stride;
        uint16* wp = (uint16*) cp0;
        tsize_t wc = cc / 2;
```

```
        if (wc > stride) {
                TIFFSwabArrayOfShort(wp, wc);
                wc -= stride;
                do {
                        REPEAT4(stride, wp[stride] += wp[0]; wp++)
                        wc -= stride;
                } while ((int32) wc > 0);
        }
}

static void
```

---

kfax'horAcc16() (./kdegraphics/kfax/libtiffax/tif_predict.c:228)

```
horAcc16(TIFF* tif, tidata_t cp0, tsize_t cc)
{
        u_int stride = PredictorState(tif)->stride;
        uint16* wp = (uint16*) cp0;
        tsize_t wc = cc / 2;

        if (wc > stride) {
                wc -= stride;
                do {
                        REPEAT4(stride, wp[stride] += wp[0]; wp++)
                        wc -= stride;
                } while ((int32) wc > 0);
        }
}

/*
 * Decode a scanline and apply the predictor routine.
 */
static int
```

---

kfax'PredictorDecodeRow() (./kdegraphics/kfax/libtiffax/tif_predict.c:247)

```
PredictorDecodeRow(TIFF* tif, tidata_t op0, tsize_t occ0, tsample_t s)
{
        TIFFPredictorState *sp = PredictorState(tif);

        assert(sp != NULL);
        assert(sp->coderow != NULL);
        assert(sp->pfunc != NULL);
        if ((*sp->coderow)(tif, op0, occ0, s)) {
                (*sp->pfunc)(tif, op0, occ0);
                return (1);
        } else
                return (0);
}

/*
 * Decode a tile/strip and apply the predictor routine.
 * Note that horizontal differencing must be done on a
 * row-by-row basis.  The width of a "row" has already
 * been calculated at pre-decode time according to the
```

```
 * strip/tile dimensions.
 */
static int
```

---

kfax'PredictorDecodeTile() (./kdegraphics/kfax/libtiffax/tif_predict.c:269)

```
PredictorDecodeTile(TIFF* tif, tidata_t op0, tsize_t occ0, tsample_t s)
{
        TIFFPredictorState *sp = PredictorState(tif);

        assert(sp != NULL);
        assert(sp->codetile != NULL);
        if ((*sp->codetile)(tif, op0, occ0, s)) {
                tsize_t rowsize = sp->rowsize;
                assert(rowsize > 0);
                assert(sp->pfunc != NULL);
                while ((long)occ0 > 0) {
                        (*sp->pfunc)(tif, op0, (tsize_t) rowsize);
                        occ0 -= rowsize;
                        op0 += rowsize;
                }
                return (1);
        } else
                return (0);
}

static void
```

---

kfax'horDiff8() (./kdegraphics/kfax/libtiffax/tif_predict.c:290)

```
horDiff8(TIFF* tif, tidata_t cp0, tsize_t cc)
{
        TIFFPredictorState* sp = PredictorState(tif);
        u_int stride = sp->stride;
        char* cp = (char*) cp0;

        if (cc > stride) {
                cc -= stride;
                /*
                 * Pipeline the most common cases.
                 */
                if (stride == 3) {
                        int r1, g1, b1;
                        int r2 = cp[0];
                        int g2 = cp[1];
                        int b2 = cp[2];
                        do {
                                r1 = cp[3]; cp[3] = r1-r2; r2 = r1;
                                g1 = cp[4]; cp[4] = g1-g2; g2 = g1;
                                b1 = cp[5]; cp[5] = b1-b2; b2 = b1;
                                cp += 3;
                        } while ((int32)(cc -= 3) > 0);
                } else if (stride == 4) {
                        int r1, g1, b1, a1;
                        int r2 = cp[0];
                        int g2 = cp[1];
```

```
                        int b2 = cp[2];
                        int a2 = cp[3];
                        do {
                                r1 = cp[4]; cp[4] = r1-r2; r2 = r1;
                                g1 = cp[5]; cp[5] = g1-g2; g2 = g1;
                                b1 = cp[6]; cp[6] = b1-b2; b2 = b1;
                                a1 = cp[7]; cp[7] = a1-a2; a2 = a1;
                                cp += 4;
                        } while ((int32)(cc -= 4) > 0);
                } else {
                        cp += cc - 1;
                        do {
                                REPEAT4(stride, cp[stride] -= cp[0]; cp--)
                        } while ((int32)(cc -= stride) > 0);
                }
        }
}

static void
```

---

kfax'horDiff16() (./kdegraphics/kfax/libtiffax/tif_predict.c:335)

```
horDiff16(TIFF* tif, tidata_t cp0, tsize_t cc)
{
        TIFFPredictorState* sp = PredictorState(tif);
        u_int stride = sp->stride;
        int16 *wp = (int16*) cp0;
        tsize_t wc = cc/2;

        if (wc > stride) {
                wc -= stride;
                wp += wc - 1;
                do {
                        REPEAT4(stride, wp[stride] -= wp[0]; wp--)
                        wc -= stride;
                } while ((int32) wc > 0);
        }
}

static int
```

---

kfax'PredictorEncodeRow() (./kdegraphics/kfax/libtiffax/tif_predict.c:353)

```
PredictorEncodeRow(TIFF* tif, tidata_t bp, tsize_t cc, tsample_t s)
{
        TIFFPredictorState *sp = PredictorState(tif);

        assert(sp != NULL);
        assert(sp->pfunc != NULL);
        assert(sp->coderow != NULL);
/* XXX horizontal differencing alters user's data XXX */
        (*sp->pfunc)(tif, bp, cc);
        return ((*sp->coderow)(tif, bp, cc, s));
}

static int
```

kfax'PredictorEncodeTile() (./kdegraphics/kfax/libtiffax/tif_predict.c:366)

```
PredictorEncodeTile(TIFF* tif, tidata_t bp0, tsize_t cc0, tsample_t s)
{
        TIFFPredictorState *sp = PredictorState(tif);
        tsize_t cc = cc0, rowsize;
        u_char* bp = bp0;

        assert(sp != NULL);
        assert(sp->pfunc != NULL);
        assert(sp->codetile != NULL);
        rowsize = sp->rowsize;
        assert(rowsize > 0);
        while ((long)cc > 0) {
                (*sp->pfunc)(tif, bp, (tsize_t) rowsize);
                cc -= rowsize;
                bp += rowsize;
        }
        return ((*sp->codetile)(tif, bp0, cc0, s));
}
```

kfax'PredictorVSetField() (./kdegraphics/kfax/libtiffax/tif_predict.c:394)

```
PredictorVSetField(TIFF* tif, ttag_t tag, va_list ap)
{
        TIFFPredictorState *sp = PredictorState(tif);

        switch (tag) {
        case TIFFTAG_PREDICTOR:
                sp->predictor = (uint16) va_arg(ap, int);
                TIFFSetFieldBit(tif, FIELD_PREDICTOR);
                break;
        default:
                return (*sp->vsetparent)(tif, tag, ap);
        }
        tif->tif_flags |= TIFF_DIRTYDIRECT;
        return (1);
}

static int
```

kfax'PredictorVGetField() (./kdegraphics/kfax/libtiffax/tif_predict.c:411)

```
PredictorVGetField(TIFF* tif, ttag_t tag, va_list ap)
{
        TIFFPredictorState *sp = PredictorState(tif);

        switch (tag) {
        case TIFFTAG_PREDICTOR:
                *va_arg(ap, uint16*) = sp->predictor;
                break;
```

```
        default:
                return (*sp->vgetparent)(tif, tag, ap);
        }
        return (1);
}

static void
```

---

kfax'PredictorPrintDir() (./kdegraphics/kfax/libtiffax/tif_predict.c:426)

```
PredictorPrintDir(TIFF* tif, FILE* fd, long flags)
{
        TIFFPredictorState* sp = PredictorState(tif);

        (void) flags;
        if (TIFFFieldSet(tif,FIELD_PREDICTOR)) {
                fprintf(fd, "  Predictor: ");
                switch (sp->predictor) {
                case 1: fprintf(fd, "none "); break;
                case 2: fprintf(fd, "horizontal differencing "); break;
                }
                fprintf(fd, "%u (0x%x)\n", sp->predictor, sp->predictor);
        }
        if (sp->printdir)
                (*sp->printdir)(tif, fd, flags);
}

int
```

---

kfax'TIFFPredictorInit() (./kdegraphics/kfax/libtiffax/tif_predict.c:444)

```
TIFFPredictorInit(TIFF* tif)
{
        TIFFPredictorState* sp = PredictorState(tif);

        /*
         * Merge codec-specific tag information and
         * override parent get/set field methods.
         */
        _TIFFMergeFieldInfo(tif, predictFieldInfo, N(predictFieldInfo));
        sp->vgetparent = tif->tif_vgetfield;
        tif->tif_vgetfield = PredictorVGetField;/* hook for predictor tag */
        sp->vsetparent = tif->tif_vsetfield;
        tif->tif_vsetfield = PredictorVSetField;/* hook for predictor tag */
        sp->printdir = tif->tif_printdir;
        tif->tif_printdir = PredictorPrintDir;  /* hook for predictor tag */

        sp->setupdecode = tif->tif_setupdecode;
        tif->tif_setupdecode = PredictorSetupDecode;
        sp->setupencode = tif->tif_setupencode;
        tif->tif_setupencode = PredictorSetupEncode;

        sp->predictor = 1;                      /* default value */
        sp->pfunc = NULL;                       /* no predictor routine */
        return (1);
}
```

kfax'TIFFPrintDirectory() (./kdegraphics/kfax/libtiffax/tif_print.c:68)

```
TIFFPrintDirectory(TIFF* tif, FILE* fd, long flags)
{
	register TIFFDirectory *td;
	char *sep;
	uint16 i;
	long l, n;

	fprintf(fd, "TIFF Directory at offset 0x%lx\n", (long) tif->tif_diroff);
	td = &tif->tif_dir;
	if (TIFFFieldSet(tif,FIELD_SUBFILETYPE)) {
		fprintf(fd, "  Subfile Type:");
		sep = " ";
		if (td->td_subfiletype & FILETYPE_REDUCEDIMAGE) {
			fprintf(fd, "%sreduced-resolution image", sep);
			sep = "/";
		}
		if (td->td_subfiletype & FILETYPE_PAGE) {
			fprintf(fd, "%smulti-page document", sep);
			sep = "/";
		}
		if (td->td_subfiletype & FILETYPE_MASK)
			fprintf(fd, "%stransparency mask", sep);
		fprintf(fd, " (%lu = 0x%lx)\n",
		    (long) td->td_subfiletype, (long) td->td_subfiletype);
	}
	if (TIFFFieldSet(tif,FIELD_IMAGEDIMENSIONS)) {
		fprintf(fd, "  Image Width: %lu Image Length: %lu",
		    (u_long) td->td_imagewidth, (u_long) td->td_imagelength);
		if (TIFFFieldSet(tif,FIELD_IMAGEDEPTH))
			fprintf(fd, " Image Depth: %lu",
			    (u_long) td->td_imagedepth);
		fprintf(fd, "\n");
	}
	if (TIFFFieldSet(tif,FIELD_TILEDIMENSIONS)) {
		fprintf(fd, "  Tile Width: %lu Tile Length: %lu",
		    (u_long) td->td_tilewidth, (u_long) td->td_tilelength);
		if (TIFFFieldSet(tif,FIELD_TILEDEPTH))
			fprintf(fd, " Tile Depth: %lu",
			    (u_long) td->td_tiledepth);
		fprintf(fd, "\n");
	}
	if (TIFFFieldSet(tif,FIELD_RESOLUTION)) {
		fprintf(fd, "  Resolution: %g, %g",
		    td->td_xresolution, td->td_yresolution);
		if (TIFFFieldSet(tif,FIELD_RESOLUTIONUNIT)) {
			switch (td->td_resolutionunit) {
			case RESUNIT_NONE:
				fprintf(fd, " (unitless)");
				break;
			case RESUNIT_INCH:
				fprintf(fd, " pixels/inch");
				break;
			case RESUNIT_CENTIMETER:
				fprintf(fd, " pixels/cm");
				break;
```

```
                default:
                        fprintf(fd, " (unit %u = 0x%x)",
                            td->td_resolutionunit,
                            td->td_resolutionunit);
                        break;
                }
        }
        fprintf(fd, "\n");
}
if (TIFFFieldSet(tif,FIELD_POSITION))
        fprintf(fd, "  Position: %g, %g\n",
            td->td_xposition, td->td_yposition);
if (TIFFFieldSet(tif,FIELD_BITSPERSAMPLE))
        fprintf(fd, "  Bits/Sample: %u\n", td->td_bitspersample);
if (TIFFFieldSet(tif,FIELD_SAMPLEFORMAT)) {
        fprintf(fd, "  Sample Format: ");
        switch (td->td_sampleformat) {
        case SAMPLEFORMAT_VOID:
                fprintf(fd, "void\n");
                break;
        case SAMPLEFORMAT_INT:
                fprintf(fd, "signed integer\n");
                break;
        case SAMPLEFORMAT_UINT:
                fprintf(fd, "unsigned integer\n");
                break;
        case SAMPLEFORMAT_IEEEFP:
                fprintf(fd, "IEEE floating point\n");
                break;
        default:
                fprintf(fd, "%u (0x%x)\n",
                    td->td_sampleformat, td->td_sampleformat);
                break;
        }
}
if (TIFFFieldSet(tif,FIELD_COMPRESSION)) {
        fprintf(fd, "  Compression Scheme: ");
        switch (td->td_compression) {
        case COMPRESSION_NONE:
                fprintf(fd, "none\n");
                break;
        case COMPRESSION_CCITTRLE:
                fprintf(fd, "CCITT modified Huffman encoding\n");
                break;
        case COMPRESSION_CCITTFAX3:
                fprintf(fd, "CCITT Group 3 facsimile encoding\n");
                break;
        case COMPRESSION_CCITTFAX4:
                fprintf(fd, "CCITT Group 4 facsimile encoding\n");
                break;
        case COMPRESSION_CCITTRLEW:
                fprintf(fd, "CCITT modified Huffman encoding %s\n",
                    "w/ word alignment");
                break;
        case COMPRESSION_PACKBITS:
                fprintf(fd, "Macintosh PackBits encoding\n");
                break;
        case COMPRESSION_THUNDERSCAN:
                fprintf(fd, "ThunderScan 4-bit encoding\n");
                break;
        case COMPRESSION_LZW:
```

```
				fprintf(fd, "Lempel-Ziv & Welch encoding\n");
				break;
			case COMPRESSION_NEXT:
				fprintf(fd, "NeXT 2-bit encoding\n");
				break;
			case COMPRESSION_OJPEG:
				fprintf(fd, "Old-style JPEG encoding\n");
				break;
			case COMPRESSION_JPEG:
				fprintf(fd, "JPEG encoding\n");
				break;
			case COMPRESSION_JBIG:
				fprintf(fd, "JBIG encoding\n");
				break;
			case COMPRESSION_DEFLATE:
				fprintf(fd, "Deflate encoding (experimental)\n");
				break;
			default:
				fprintf(fd, "%u (0x%x)\n",
				    td->td_compression, td->td_compression);
				break;
			}
		}
		if (TIFFFieldSet(tif,FIELD_PHOTOMETRIC)) {
			fprintf(fd, "  Photometric Interpretation: ");
			if (td->td_photometric < NPHOTONAMES)
				fprintf(fd, "%s\n", photoNames[td->td_photometric]);
			else
				fprintf(fd, "%u (0x%x)\n",
				    td->td_photometric, td->td_photometric);
		}
		if (TIFFFieldSet(tif,FIELD_EXTRASAMPLES) && td->td_extrasamples) {
			fprintf(fd, "  Extra Samples: %u<", td->td_extrasamples);
			sep = "";
			for (i = 0; i < td->td_extrasamples; i++) {
				switch (td->td_sampleinfo[i]) {
				case EXTRASAMPLE_UNSPECIFIED:
					fprintf(fd, "%sunspecified", sep);
					break;
				case EXTRASAMPLE_ASSOCALPHA:
					fprintf(fd, "%sassoc-alpha", sep);
					break;
				case EXTRASAMPLE_UNASSALPHA:
					fprintf(fd, "%sunassoc-alpha", sep);
					break;
				default:
					fprintf(fd, "%s%u (0x%x)", sep,
					    td->td_sampleinfo[i], td->td_sampleinfo[i]);
					break;
				}
				sep = ", ";
			}
			fprintf(fd, ">\n");
		}
#ifdef CMYK_SUPPORT
		if (TIFFFieldSet(tif,FIELD_INKSET)) {
			fprintf(fd, "  Ink Set: ");
			switch (td->td_inkset) {
			case INKSET_CMYK:
				fprintf(fd, "CMYK\n");
				break;
```

```
                default:
                        fprintf(fd, "%u (0x%x)\n",
                            td->td_inkset, td->td_inkset);
                        break;
                }
        }
        if (TIFFFieldSet(tif,FIELD_INKNAMES)) {
                char* cp;
                fprintf(fd, "  Ink Names: ");
                i = td->td_samplesperpixel;
                sep = "";
                for (cp = td->td_inknames; i > 0; cp = strchr(cp, '\0')) {
                        fprintf(fd, "%s", sep);
                        _TIFFprintAscii(fd, cp);
                        sep = ", ";
                }
        }
        if (TIFFFieldSet(tif,FIELD_DOTRANGE))
                fprintf(fd, "  Dot Range: %u-%u\n",
                    td->td_dotrange[0], td->td_dotrange[1]);
        if (TIFFFieldSet(tif,FIELD_TARGETPRINTER))
                _TIFFprintAsciiTag(fd, "Target Printer", td->td_targetprinter);
#endif
        if (TIFFFieldSet(tif,FIELD_THRESHHOLDING)) {
                fprintf(fd, "  Thresholding: ");
                switch (td->td_threshholding) {
                case THRESHHOLD_BILEVEL:
                        fprintf(fd, "bilevel art scan\n");
                        break;
                case THRESHHOLD_HALFTONE:
                        fprintf(fd, "halftone or dithered scan\n");
                        break;
                case THRESHHOLD_ERRORDIFFUSE:
                        fprintf(fd, "error diffused\n");
                        break;
                default:
                        fprintf(fd, "%u (0x%x)\n",
                            td->td_threshholding, td->td_threshholding);
                        break;
                }
        }
        if (TIFFFieldSet(tif,FIELD_FILLORDER)) {
                fprintf(fd, "  FillOrder: ");
                switch (td->td_fillorder) {
                case FILLORDER_MSB2LSB:
                        fprintf(fd, "msb-to-lsb\n");
                        break;
                case FILLORDER_LSB2MSB:
                        fprintf(fd, "lsb-to-msb\n");
                        break;
                default:
                        fprintf(fd, "%u (0x%x)\n",
                            td->td_fillorder, td->td_fillorder);
                        break;
                }
        }
#ifdef YCBCR_SUPPORT
        if (TIFFFieldSet(tif,FIELD_YCBCRSUBSAMPLING))
                fprintf(fd, "  YCbCr Subsampling: %u, %u\n",
                    td->td_ycbcrsubsampling[0], td->td_ycbcrsubsampling[1]);
        if (TIFFFieldSet(tif,FIELD_YCBCRPOSITIONING)) {
```

```
		fprintf(fd, "  YCbCr Positioning: ");
		switch (td->td_ycbcrpositioning) {
		case YCBCRPOSITION_CENTERED:
			fprintf(fd, "centered\n");
			break;
		case YCBCRPOSITION_COSITED:
			fprintf(fd, "cosited\n");
			break;
		default:
			fprintf(fd, "%u (0x%x)\n",
			    td->td_ycbcrpositioning, td->td_ycbcrpositioning);
			break;
		}
	}
	if (TIFFFieldSet(tif,FIELD_YCBCRCOEFFICIENTS))
		fprintf(fd, "  YCbCr Coefficients: %g, %g, %g\n",
		    td->td_ycbcrcoeffs[0],
		    td->td_ycbcrcoeffs[1],
		    td->td_ycbcrcoeffs[2]);
#endif
	if (TIFFFieldSet(tif,FIELD_HALFTONEHINTS))
		fprintf(fd, "  Halftone Hints: light %u dark %u\n",
		    td->td_halftonehints[0], td->td_halftonehints[1]);
	if (TIFFFieldSet(tif,FIELD_ARTIST))
		_TIFFprintAsciiTag(fd, "Artist", td->td_artist);
	if (TIFFFieldSet(tif,FIELD_DATETIME))
		_TIFFprintAsciiTag(fd, "Date & Time", td->td_datetime);
	if (TIFFFieldSet(tif,FIELD_HOSTCOMPUTER))
		_TIFFprintAsciiTag(fd, "Host Computer", td->td_hostcomputer);
	if (TIFFFieldSet(tif,FIELD_SOFTWARE))
		_TIFFprintAsciiTag(fd, "Software", td->td_software);
	if (TIFFFieldSet(tif,FIELD_DOCUMENTNAME))
		_TIFFprintAsciiTag(fd, "Document Name", td->td_documentname);
	if (TIFFFieldSet(tif,FIELD_IMAGEDESCRIPTION))
		_TIFFprintAsciiTag(fd, "Image Description", td->td_imagedescript
	if (TIFFFieldSet(tif,FIELD_MAKE))
		_TIFFprintAsciiTag(fd, "Make", td->td_make);
	if (TIFFFieldSet(tif,FIELD_MODEL))
		_TIFFprintAsciiTag(fd, "Model", td->td_model);
	if (TIFFFieldSet(tif,FIELD_ORIENTATION)) {
		fprintf(fd, "  Orientation: ");
		if (td->td_orientation < NORIENTNAMES)
			fprintf(fd, "%s\n", orientNames[td->td_orientation]);
		else
			fprintf(fd, "%u (0x%x)\n",
			    td->td_orientation, td->td_orientation);
	}
	if (TIFFFieldSet(tif,FIELD_SAMPLESPERPIXEL))
		fprintf(fd, "  Samples/Pixel: %u\n", td->td_samplesperpixel);
	if (TIFFFieldSet(tif,FIELD_ROWSPERSTRIP)) {
		fprintf(fd, "  Rows/Strip: ");
		if (td->td_rowsperstrip == (uint32) -1)
			fprintf(fd, "(infinite)\n");
		else
			fprintf(fd, "%lu\n", (u_long) td->td_rowsperstrip);
	}
	if (TIFFFieldSet(tif,FIELD_MINSAMPLEVALUE))
		fprintf(fd, "  Min Sample Value: %u\n", td->td_minsamplevalue);
	if (TIFFFieldSet(tif,FIELD_MAXSAMPLEVALUE))
		fprintf(fd, "  Max Sample Value: %u\n", td->td_maxsamplevalue);
	if (TIFFFieldSet(tif,FIELD_SMINSAMPLEVALUE))
```

```
                fprintf(fd, "  SMin Sample Value: %g\n",
                    td->td_sminsamplevalue);
        if (TIFFFieldSet(tif,FIELD_SMAXSAMPLEVALUE))
                fprintf(fd, "  SMax Sample Value: %g\n",
                    td->td_smaxsamplevalue);
        if (TIFFFieldSet(tif,FIELD_PLANARCONFIG)) {
                fprintf(fd, "  Planar Configuration: ");
                switch (td->td_planarconfig) {
                case PLANARCONFIG_CONTIG:
                        fprintf(fd, "single image plane\n");
                        break;
                case PLANARCONFIG_SEPARATE:
                        fprintf(fd, "separate image planes\n");
                        break;
                default:
                        fprintf(fd, "%u (0x%x)\n",
                            td->td_planarconfig, td->td_planarconfig);
                        break;
                }
        }
        if (TIFFFieldSet(tif,FIELD_PAGENAME))
                _TIFFprintAsciiTag(fd, "Page Name", td->td_pagename);
        if (TIFFFieldSet(tif,FIELD_PAGENUMBER))
                fprintf(fd, "  Page Number: %u-%u\n",
                    td->td_pagenumber[0], td->td_pagenumber[1]);
        if (TIFFFieldSet(tif,FIELD_COLORMAP)) {
                fprintf(fd, "  Color Map: ");
                if (flags & TIFFPRINT_COLORMAP) {
                        fprintf(fd, "\n");
                        n = 1L<<td->td_bitspersample;
                        for (l = 0; l < n; l++)
                                fprintf(fd, "   %5lu: %5u %5u %5u\n",
                                    l,
                                    td->td_colormap[0][l],
                                    td->td_colormap[1][l],
                                    td->td_colormap[2][l]);
                } else
                        fprintf(fd, "(present)\n");
        }
#ifdef COLORIMETRY_SUPPORT
        if (TIFFFieldSet(tif,FIELD_WHITEPOINT))
                fprintf(fd, "  White Point: %g-%g\n",
                    td->td_whitepoint[0], td->td_whitepoint[1]);
        if (TIFFFieldSet(tif,FIELD_PRIMARYCHROMAS))
                fprintf(fd, "  Primary Chromaticities: %g,%g %g,%g %g,%g\n",
                    td->td_primarychromas[0], td->td_primarychromas[1],
                    td->td_primarychromas[2], td->td_primarychromas[3],
                    td->td_primarychromas[4], td->td_primarychromas[5]);
        if (TIFFFieldSet(tif,FIELD_REFBLACKWHITE)) {
                fprintf(fd, "  Reference Black/White:\n");
                for (i = 0; i < td->td_samplesperpixel; i++)
                        fprintf(fd, "    %2d: %5g %5g\n",
                            i,
                            td->td_refblackwhite[2*i+0],
                            td->td_refblackwhite[2*i+1]);
        }
        if (TIFFFieldSet(tif,FIELD_TRANSFERFUNCTION)) {
                fprintf(fd, "  Transfer Function: ");
                if (flags & TIFFPRINT_CURVES) {
                        fprintf(fd, "\n");
                        n = 1L<<td->td_bitspersample;
```

```
                        for (l = 0; l < n; l++) {
                                fprintf(fd, "    %2lu: %5u",
                                    l, td->td_transferfunction[0][l]);
                                for (i = 1; i < td->td_samplesperpixel; i++)
                                        fprintf(fd, " %5u",
                                            td->td_transferfunction[i][l]);
                                fputc('\n', fd);
                        }
                } else
                        fprintf(fd, "(present)\n");
        }
#endif
#if SUBIFD_SUPPORT
        if (TIFFFieldSet(tif, FIELD_SUBIFD)) {
                fprintf(fd, "  SubIFD Offsets:");
                for (i = 0; i < td->td_nsubifd; i++)
                        fprintf(fd, " %5lu", (long) td->td_subifd[i]);
                fputc('\n', fd);
        }
#endif
        if (tif->tif_printdir)
                (*tif->tif_printdir)(tif, fd, flags);
        if ((flags & TIFFPRINT_STRIPS) &&
            TIFFFieldSet(tif,FIELD_STRIPOFFSETS)) {
                tstrip_t s;

                fprintf(fd, "  %lu %s:\n",
                    (long) td->td_nstrips,
                    isTiled(tif) ? "Tiles" : "Strips");
                for (s = 0; s < td->td_nstrips; s++)
                        fprintf(fd, "    %3lu: [%8lu, %8lu]\n",
                            (u_long) s,
                            (u_long) td->td_stripoffset[s],
                            (u_long) td->td_stripbytecount[s]);
        }
}

void
```

---

kfax'_TIFFprintAscii() (./kdegraphics/kfax/libtiffax/tif_print.c:466)

```
_TIFFprintAscii(FILE* fd, const char* cp)
{
        for (; *cp != '\0'; cp++) {
                const char* tp;

                if (isprint(*cp)) {
                        fputc(*cp, fd);
                        continue;
                }
                for (tp = "\tt\bb\rr\nn\vv"; *tp; tp++)
                        if (*tp++ == *cp)
                                break;
                if (*tp)
                        fprintf(fd, "\\%c", *tp);
                else
                        fprintf(fd, "\\%03o", *cp & 0xff);
        }
```

```
}

void
```

---

kfax'_TIFFprintAsciiTag() (./kdegraphics/kfax/libtiffax/tif_print.c:486)

```
_TIFFprintAsciiTag(FILE* fd, const char* name, const char* value)
{
        fprintf(fd, "  %s: \"", name);
        _TIFFprintAscii(fd, value);
        fprintf(fd, "\"\n");
}
```

---

kfax'TIFFSeek() (./kdegraphics/kfax/libtiffax/tif_read.c:48)

```
TIFFSeek(TIFF* tif, uint32 row, tsample_t sample)
{
        register TIFFDirectory *td = &tif->tif_dir;
        tstrip_t strip;

        if (row >= td->td_imagelength) {        /* out of range */
                TIFFError(tif->tif_name, "%lu: Row out of range, max %lu",
                    (u_long) row, (u_long) td->td_imagelength);
                return (0);
        }
        if (td->td_planarconfig == PLANARCONFIG_SEPARATE) {
                if (sample >= td->td_samplesperpixel) {
                        TIFFError(tif->tif_name,
                            "%lu: Sample out of range, max %lu",
                            (u_long) sample, (u_long) td->td_samplesperpixel);
                        return (0);
                }
                strip = sample*td->td_stripsperimage + row/td->td_rowsperstrip;
        } else
                strip = row / td->td_rowsperstrip;
        if (strip != tif->tif_curstrip) {       /* different strip, refill */
                if (!TIFFFillStrip(tif, strip))
                        return (0);
        } else if (row < tif->tif_row) {
                /*
                 * Moving backwards within the same strip: backup
                 * to the start and then decode forward (below).
                 *
                 * NB: If you're planning on lots of random access within a
                 * strip, it's better to just read and decode the entire
                 * strip, and then access the decoded data in a random fashion.
                 */
                if (!TIFFStartStrip(tif, strip))
                        return (0);
        }
        if (row != tif->tif_row) {
                /*
                 * Seek forward to the desired row.
                 */
                if (!(*tif->tif_seek)(tif, row - tif->tif_row))
                        return (0);
```

```
                tif->tif_row = row;
        }
        return (1);
}

int
```

---

kfax'TIFFReadScanline() (./kdegraphics/kfax/libtiffax/tif_read.c:95)

```
TIFFReadScanline(TIFF* tif, tdata_t buf, uint32 row, tsample_t sample)
{
        int e;

        if (!TIFFCheckRead(tif, 0))
                return (-1);
        if (e = TIFFSeek(tif, row, sample)) {
                /*
                 * Decompress desired row into user buffer.
                 */
                e = (*tif->tif_decoderow)
                    (tif, (tidata_t) buf, tif->tif_scanlinesize, sample);
                tif->tif_row++;
                if (e)
                        (*tif->tif_postdecode)(tif, (tidata_t) buf,
                            tif->tif_scanlinesize);
        }
        return (e ? 1 : -1);
}

/*
 * Read a strip of data and decompress the specified
 * amount into the user-supplied buffer.
 */
tsize_t
```

---

kfax'TIFFReadEncodedStrip() (./kdegraphics/kfax/libtiffax/tif_read.c:120)

```
TIFFReadEncodedStrip(TIFF* tif, tstrip_t strip, tdata_t buf, tsize_t size)
{
        TIFFDirectory *td = &tif->tif_dir;
        uint32 nrows;
        tsize_t stripsize;

        if (!TIFFCheckRead(tif, 0))
                return (-1);
        if (strip >= td->td_nstrips) {
                TIFFError(tif->tif_name, "%ld: Strip out of range, max %ld",
                    (long) strip, (long) td->td_nstrips);
                return (-1);
        }
        /*
         * Calculate the strip size according to the number of
         * rows in the strip (check for truncated last strip).
         */
        if (strip != td->td_nstrips-1 ||
            (nrows = td->td_imagelength % td->td_rowsperstrip) == 0)
```

```
                nrows = td->td_rowsperstrip;
        stripsize = TIFFVStripSize(tif, nrows);
        if (size == (tsize_t) -1)
                size = stripsize;
        else if (size > stripsize)
                size = stripsize;
        if (TIFFFillStrip(tif, strip) && (*tif->tif_decodestrip)(tif,
            (tidata_t) buf, size, (tsample_t)(strip / td->td_stripsperimage))) {
                (*tif->tif_postdecode)(tif, (tidata_t) buf, size);
                return (size);
        } else
                return ((tsize_t) -1);
}

static tsize_t
```

---

kfax'TIFFReadRawStrip1() (./kdegraphics/kfax/libtiffax/tif_read.c:154)

```
TIFFReadRawStrip1(TIFF* tif,
    tstrip_t strip, tdata_t buf, tsize_t size, const char* module)
{
        TIFFDirectory *td = &tif->tif_dir;

        if (!isMapped(tif)) {
                if (!SeekOK(tif, td->td_stripoffset[strip])) {
                        TIFFError(module,
                            "%s: Seek error at scanline %lu, strip %lu",
                            tif->tif_name,
                            (u_long) tif->tif_row, (u_long) strip);
                        return (-1);
                }
                if (!ReadOK(tif, buf, size)) {
                        TIFFError(module, "%s: Read error at scanline %lu",
                            tif->tif_name, (u_long) tif->tif_row);
                        return (-1);
                }
        } else {
                if (td->td_stripoffset[strip] + size > tif->tif_size) {
                        TIFFError(module,
                            "%s: Seek error at scanline %lu, strip %lu",
                            tif->tif_name,
                            (u_long) tif->tif_row, (u_long) strip);
                        return (-1);
                }
                _TIFFmemcpy(buf, tif->tif_base + td->td_stripoffset[strip], size
        }
        return (size);
}

/*
 * Read a strip of data from the file.
 */
tsize_t
```

---

kfax'TIFFReadRawStrip() (./kdegraphics/kfax/libtiffax/tif_read.c:189)

```
TIFFReadRawStrip(TIFF* tif, tstrip_t strip, tdata_t buf, tsize_t size)
{
        static const char module[] = "TIFFReadRawStrip";
        TIFFDirectory *td = &tif->tif_dir;
        tsize_t bytecount;

        if (!TIFFCheckRead(tif, 0))
                return ((tsize_t) -1);
        if (strip >= td->td_nstrips) {
                TIFFError(tif->tif_name, "%lu: Strip out of range, max %lu",
                    (u_long) strip, (u_long) td->td_nstrips);
                return ((tsize_t) -1);
        }
        bytecount = td->td_stripbytecount[strip];
        if (bytecount <= 0) {
                TIFFError(tif->tif_name,
                    "%lu: Invalid strip byte count, strip %lu",
                    (u_long) bytecount, (u_long) strip);
                return ((tsize_t) -1);
        }
        if (size != (tsize_t)-1 && size < bytecount)
                bytecount = size;
        return (TIFFReadRawStrip1(tif, strip, buf, bytecount, module));
}

/*
 * Read the specified strip and setup for decoding.
 * The data buffer is expanded, as necessary, to
 * hold the strip's data.
 */
static int
```

---

kfax'TIFFFillStrip() (./kdegraphics/kfax/libtiffax/tif_read.c:220)

```
TIFFFillStrip(TIFF* tif, tstrip_t strip)
{
        static const char module[] = "TIFFFillStrip";
        TIFFDirectory *td = &tif->tif_dir;
        tsize_t bytecount;

        bytecount = td->td_stripbytecount[strip];
        if (bytecount <= 0) {
                TIFFError(tif->tif_name,
                    "%lu: Invalid strip byte count, strip %lu",
                    (u_long) bytecount, (u_long) strip);
                return (0);
        }
        if (isMapped(tif) &&
            (isFillOrder(tif, td->td_fillorder) || (tif->tif_flags & TIFF_NOBITRE
                /*
                 * The image is mapped into memory and we either don't
                 * need to flip bits or the compression routine is going
                 * to handle this operation itself.  In this case, avoid
                 * copying the raw data and instead just reference the
                 * data from the memory mapped file image.  This assumes
                 * that the decompression routines do not modify the
                 * contents of the raw data buffer (if they try to,
```

```
                 * the application will get a fault since the file is
                 * mapped read-only).
                 */
                if ((tif->tif_flags & TIFF_MYBUFFER) && tif->tif_rawdata)
                        _TIFFfree(tif->tif_rawdata);
                tif->tif_flags &= ~TIFF_MYBUFFER;
                if (td->td_stripoffset[strip] + bytecount > tif->tif_size) {
                        /*
                         * This error message might seem strange, but it's
                         * what would happen if a read were done instead.
                         */
                        TIFFError(module, "%s: Read error on strip %lu",
                            tif->tif_name, (u_long) strip);
                        tif->tif_curstrip = NOSTRIP;
                        return (0);
                }
                tif->tif_rawdatasize = bytecount;
                tif->tif_rawdata = tif->tif_base + td->td_stripoffset[strip];
        } else {
                /*
                 * Expand raw data buffer, if needed, to
                 * hold data strip coming from file
                 * (perhaps should set upper bound on
                 *  the size of a buffer we'll use?).
                 */
                if (bytecount > tif->tif_rawdatasize) {
                        tif->tif_curstrip = NOSTRIP;
                        if ((tif->tif_flags & TIFF_MYBUFFER) == 0) {
                                TIFFError(module,
                                "%s: Data buffer too small to hold strip %lu",
                                    tif->tif_name, (u_long) strip);
                                return (0);
                        }
                        if (!TIFFReadBufferSetup(tif, 0,
                            TIFFroundup(bytecount, 1024)))
                                return (0);
                }
                if (TIFFReadRawStrip1(tif, strip, (u_char *)tif->tif_rawdata,
                    bytecount, module) != bytecount)
                        return (0);
                if (!isFillOrder(tif, td->td_fillorder) &&
                    (tif->tif_flags & TIFF_NOBITREV) == 0)
                        TIFFReverseBits(tif->tif_rawdata, bytecount);
        }
        return (TIFFStartStrip(tif, strip));
}

/*
 * Tile-oriented Read Support
 * Contributed by Nancy Cam (Silicon Graphics).
 */

/*
 * Read and decompress a tile of data.  The
 * tile is selected by the (x,y,z,s) coordinates.
 */
tsize_t
```

kfax'TIFFReadTile() (./kdegraphics/kfax/libtiffax/tif_read.c:300)

```
TIFFReadTile(TIFF* tif,
    tdata_t buf, uint32 x, uint32 y, uint32 z, tsample_t s)
{
        if (!TIFFCheckRead(tif, 1) || !TIFFCheckTile(tif, x, y, z, s))
                return (-1);
        return (TIFFReadEncodedTile(tif,
            TIFFComputeTile(tif, x, y, z, s), buf, (tsize_t) -1));
}

/*
 * Read a tile of data and decompress the specified
 * amount into the user-supplied buffer.
 */
tsize_t
```

---

kfax'TIFFReadEncodedTile() (./kdegraphics/kfax/libtiffax/tif_read.c:314)

```
TIFFReadEncodedTile(TIFF* tif, ttile_t tile, tdata_t buf, tsize_t size)
{
        TIFFDirectory *td = &tif->tif_dir;
        tsize_t tilesize = tif->tif_tilesize;

        if (!TIFFCheckRead(tif, 1))
                return (-1);
        if (tile >= td->td_nstrips) {
                TIFFError(tif->tif_name, "%ld: Tile out of range, max %ld",
                    (long) tile, (u_long) td->td_nstrips);
                return (-1);
        }
        if (size == (tsize_t) -1)
                size = tilesize;
        else if (size > tilesize)
                size = tilesize;
        if (TIFFFillTile(tif, tile) && (*tif->tif_decodetile)(tif,
            (tidata_t) buf, size, (tsample_t)(tile/td->td_stripsperimage))) {
                (*tif->tif_postdecode)(tif, (tidata_t) buf, size);
                return (size);
        } else
                return (-1);
}

static tsize_t
```

---

kfax'TIFFReadRawTile1() (./kdegraphics/kfax/libtiffax/tif_read.c:339)

```
TIFFReadRawTile1(TIFF* tif,
    ttile_t tile, tdata_t buf, tsize_t size, const char* module)
{
        TIFFDirectory *td = &tif->tif_dir;

        if (!isMapped(tif)) {
                if (!SeekOK(tif, td->td_stripoffset[tile])) {
```

```
                        TIFFError(module,
                            "%s: Seek error at row %ld, col %ld, tile %ld",
                            tif->tif_name,
                            (long) tif->tif_row,
                            (long) tif->tif_col,
                            (long) tile);
                        return ((tsize_t) -1);
                }
                if (!ReadOK(tif, buf, size)) {
                        TIFFError(module, "%s: Read error at row %ld, col %ld",
                            tif->tif_name,
                            (long) tif->tif_row,
                            (long) tif->tif_col);
                        return ((tsize_t) -1);
                }
        } else {
                if (td->td_stripoffset[tile] + size > tif->tif_size) {
                        TIFFError(module,
                            "%s: Seek error at row %ld, col %ld, tile %ld",
                            tif->tif_name,
                            (long) tif->tif_row,
                            (long) tif->tif_col,
                            (long) tile);
                        return ((tsize_t) -1);
                }
                _TIFFmemcpy(buf, tif->tif_base + td->td_stripoffset[tile], size)
        }
        return (size);
}

/*
 * Read a tile of data from the file.
 */
tsize_t
```

---

kfax'TIFFReadRawTile() (./kdegraphics/kfax/libtiffax/tif_read.c:380)

```
TIFFReadRawTile(TIFF* tif, ttile_t tile, tdata_t buf, tsize_t size)
{
        static const char module[] = "TIFFReadRawTile";
        TIFFDirectory *td = &tif->tif_dir;
        tsize_t bytecount;

        if (!TIFFCheckRead(tif, 1))
                return ((tsize_t) -1);
        if (tile >= td->td_nstrips) {
                TIFFError(tif->tif_name, "%lu: Tile out of range, max %lu",
                    (u_long) tile, (u_long) td->td_nstrips);
                return ((tsize_t) -1);
        }
        bytecount = td->td_stripbytecount[tile];
        if (size != (tsize_t) -1 && size < bytecount)
                bytecount = size;
        return (TIFFReadRawTile1(tif, tile, buf, bytecount, module));
}

/*
 * Read the specified tile and setup for decoding.
```

```
 * The data buffer is expanded, as necessary, to
 * hold the tile's data.
 */
static int
```

---

kfax'TIFFFillTile() (./kdegraphics/kfax/libtiffax/tif_read.c:405)

```
TIFFFillTile(TIFF* tif, ttile_t tile)
{
        static const char module[] = "TIFFFillTile";
        TIFFDirectory *td = &tif->tif_dir;
        tsize_t bytecount;

        bytecount = td->td_stripbytecount[tile];
        if (bytecount <= 0) {
                TIFFError(tif->tif_name,
                    "%lu: Invalid tile byte count, tile %lu",
                    (u_long) bytecount, (u_long) tile);
                return (0);
        }
        if (isMapped(tif) &&
            (isFillOrder(tif, td->td_fillorder) || (tif->tif_flags & TIFF_NOBITRI
                /*
                 * The image is mapped into memory and we either don't
                 * need to flip bits or the compression routine is going
                 * to handle this operation itself.  In this case, avoid
                 * copying the raw data and instead just reference the
                 * data from the memory mapped file image.  This assumes
                 * that the decompression routines do not modify the
                 * contents of the raw data buffer (if they try to,
                 * the application will get a fault since the file is
                 * mapped read-only).
                 */
                if ((tif->tif_flags & TIFF_MYBUFFER) && tif->tif_rawdata)
                        _TIFFfree(tif->tif_rawdata);
                tif->tif_flags &= ~TIFF_MYBUFFER;
                if (td->td_stripoffset[tile] + bytecount > tif->tif_size) {
                        tif->tif_curtile = NOTILE;
                        return (0);
                }
                tif->tif_rawdatasize = bytecount;
                tif->tif_rawdata = tif->tif_base + td->td_stripoffset[tile];
        } else {
                /*
                 * Expand raw data buffer, if needed, to
                 * hold data tile coming from file
                 * (perhaps should set upper bound on
                 *  the size of a buffer we'll use?).
                 */
                if (bytecount > tif->tif_rawdatasize) {
                        tif->tif_curtile = NOTILE;
                        if ((tif->tif_flags & TIFF_MYBUFFER) == 0) {
                                TIFFError(module,
                                "%s: Data buffer too small to hold tile %ld",
                                    tif->tif_name, (long) tile);
                                return (0);
                        }
                        if (!TIFFReadBufferSetup(tif, 0,
```

```
                            TIFFroundup(bytecount, 1024)))
                                return (0);
                }
                if (TIFFReadRawTile1(tif, tile, (u_char *)tif->tif_rawdata,
                    bytecount, module) != bytecount)
                        return (0);
                if (!isFillOrder(tif, td->td_fillorder) &&
                    (tif->tif_flags & TIFF_NOBITREV) == 0)
                        TIFFReverseBits(tif->tif_rawdata, bytecount);
        }
        return (TIFFStartTile(tif, tile));
}

/*
 * Setup the raw data buffer in preparation for
 * reading a strip of raw data.  If the buffer
 * is specified as zero, then a buffer of appropriate
 * size is allocated by the library.  Otherwise,
 * the client must guarantee that the buffer is
 * large enough to hold any individual strip of
 * raw data.
 */
int
```

---

kfax'TIFFReadBufferSetup() (./kdegraphics/kfax/libtiffax/tif_read.c:479)

```
TIFFReadBufferSetup(TIFF* tif, tdata_t bp, tsize_t size)
{
        static const char module[] = "TIFFReadBufferSetup";

        if (tif->tif_rawdata) {
                if (tif->tif_flags & TIFF_MYBUFFER)
                        _TIFFfree(tif->tif_rawdata);
                tif->tif_rawdata = NULL;
        }
        if (bp) {
                tif->tif_rawdatasize = size;
                tif->tif_rawdata = (tidata_t) bp;
                tif->tif_flags &= ~TIFF_MYBUFFER;
        } else {
                tif->tif_rawdatasize = TIFFroundup(size, 1024);
                tif->tif_rawdata = (tidata_t) _TIFFmalloc(tif->tif_rawdatasize);
                tif->tif_flags |= TIFF_MYBUFFER;
        }
        if (tif->tif_rawdata == NULL) {
                TIFFError(module,
                    "%s: No space for data buffer at scanline %ld",
                    tif->tif_name, (long) tif->tif_row);
                tif->tif_rawdatasize = 0;
                return (0);
        }
        return (1);
}

/*
 * Set state to appear as if a
 * strip has just been read in.
 */
```

```
static int
```

---

kfax'TIFFStartStrip() (./kdegraphics/kfax/libtiffax/tif_read.c:512)

```
TIFFStartStrip(TIFF* tif, tstrip_t strip)
{
        TIFFDirectory *td = &tif->tif_dir;

        if ((tif->tif_flags & TIFF_CODERSETUP) == 0) {
                if (!(*tif->tif_setupdecode)(tif))
                        return (0);
                tif->tif_flags |= TIFF_CODERSETUP;
        }
        tif->tif_curstrip = strip;
        tif->tif_row = (strip % td->td_stripsperimage) * td->td_rowsperstrip;
        tif->tif_rawcp = tif->tif_rawdata;
        tif->tif_rawcc = td->td_stripbytecount[strip];
        return ((*tif->tif_predecode)(tif,
                        (tsample_t)(strip / td->td_stripsperimage)));
}

/*
 * Set state to appear as if a
 * tile has just been read in.
 */
static int
```

---

kfax'TIFFStartTile() (./kdegraphics/kfax/libtiffax/tif_read.c:534)

```
TIFFStartTile(TIFF* tif, ttile_t tile)
{
        TIFFDirectory *td = &tif->tif_dir;

        if ((tif->tif_flags & TIFF_CODERSETUP) == 0) {
                if (!(*tif->tif_setupdecode)(tif))
                        return (0);
                tif->tif_flags |= TIFF_CODERSETUP;
        }
        tif->tif_curtile = tile;
        tif->tif_row =
            (tile % TIFFhowmany(td->td_imagewidth, td->td_tilewidth)) *
                td->td_tilelength;
        tif->tif_col =
            (tile % TIFFhowmany(td->td_imagelength, td->td_tilelength)) *
                td->td_tilewidth;
        tif->tif_rawcp = tif->tif_rawdata;
        tif->tif_rawcc = td->td_stripbytecount[tile];
        return ((*tif->tif_predecode)(tif,
                        (tsample_t)(tile/td->td_stripsperimage)));
}

static int
```

---

kfax'TIFFCheckRead() (./kdegraphics/kfax/libtiffax/tif_read.c:557)

```
TIFFCheckRead(TIFF* tif, int tiles)
{
        if (tif->tif_mode == O_WRONLY) {
                TIFFError(tif->tif_name, "File not open for reading");
                return (0);
        }
        if (tiles ^ isTiled(tif)) {
                TIFFError(tif->tif_name, tiles ?
                    "Can not read tiles from a stripped image" :
                    "Can not read scanlines from a tiled image");
                return (0);
        }
        return (1);
}

void
```

---

kfax'_TIFFNoPostDecode() (./kdegraphics/kfax/libtiffax/tif_read.c:573)

```
_TIFFNoPostDecode(TIFF* tif, tidata_t buf, tsize_t cc)
{
    (void) tif; (void) buf; (void) cc;
}

void
```

---

kfax'_TIFFSwab16BitData() (./kdegraphics/kfax/libtiffax/tif_read.c:579)

```
_TIFFSwab16BitData(TIFF* tif, tidata_t buf, tsize_t cc)
{
    (void) tif;
    assert((cc & 1) == 0);
    TIFFSwabArrayOfShort((uint16*) buf, cc/2);
}

void
```

---

kfax'_TIFFSwab32BitData() (./kdegraphics/kfax/libtiffax/tif_read.c:587)

```
_TIFFSwab32BitData(TIFF* tif, tidata_t buf, tsize_t cc)
{
    (void) tif;
    assert((cc & 3) == 0);
    TIFFSwabArrayOfLong((uint32*) buf, cc/4);
}

void
```

---

kfax'_TIFFSwab64BitData() (./kdegraphics/kfax/libtiffax/tif_read.c:595)

```
_TIFFSwab64BitData(TIFF* tif, tidata_t buf, tsize_t cc)
{
    (void) tif;
    assert((cc & 7) == 0);
    TIFFSwabArrayOfDouble((double*) buf, cc/8);
}
```

---

kfax'TIFFComputeStrip() (./kdegraphics/kfax/libtiffax/tif_strip.c:38)

```
TIFFComputeStrip(TIFF* tif, uint32 row, tsample_t sample)
{
        TIFFDirectory *td = &tif->tif_dir;
        tstrip_t strip;

        strip = row / td->td_rowsperstrip;
        if (td->td_planarconfig == PLANARCONFIG_SEPARATE) {
                if (sample >= td->td_samplesperpixel) {
                        TIFFError(tif->tif_name,
                            "%u: Sample out of range, max %u",
                            sample, td->td_samplesperpixel);
                        return ((tstrip_t) 0);
                }
                strip += sample*td->td_stripsperimage;
        }
        return (strip);
}

/*
 * Compute how many strips are in an image.
 */
tstrip_t
```

---

kfax'TIFFNumberOfStrips() (./kdegraphics/kfax/libtiffax/tif_strip.c:60)

```
TIFFNumberOfStrips(TIFF* tif)
{
        TIFFDirectory *td = &tif->tif_dir;
        tstrip_t nstrips;

        nstrips = (td->td_rowsperstrip == (uint32) -1 ?
             (td->td_imagelength != 0 ? 1 : 0) :
             TIFFhowmany(td->td_imagelength, td->td_rowsperstrip));
        if (td->td_planarconfig == PLANARCONFIG_SEPARATE)
                nstrips *= td->td_samplesperpixel;
        return (nstrips);
}

/*
 * Compute the # bytes in a variable height, row-aligned strip.
 */
tsize_t
```

kfax'TIFFVStripSize() (./kdegraphics/kfax/libtiffax/tif_strip.c:77)

```
TIFFVStripSize(TIFF* tif, uint32 nrows)
{
        TIFFDirectory *td = &tif->tif_dir;

        if (nrows == (uint32) -1)
                nrows = td->td_imagelength;
#ifdef YCBCR_SUPPORT
        if (td->td_planarconfig == PLANARCONFIG_CONTIG &&
            td->td_photometric == PHOTOMETRIC_YCBCR &&
            !isUpSampled(tif)) {
                /*
                 * Packed YCbCr data contain one Cb+Cr for every
                 * HorizontalSampling*VerticalSampling Y values.
                 * Must also roundup width and height when calculating
                 * since images that are not a multiple of the
                 * horizontal/vertical subsampling area include
                 * YCbCr data for the extended image.
                 */
                tsize_t w =
                    TIFFroundup(td->td_imagewidth, td->td_ycbcrsubsampling[0]);
                tsize_t scanline = TIFFhowmany(w*td->td_bitspersample, 8);
                tsize_t samplingarea =
                    td->td_ycbcrsubsampling[0]*td->td_ycbcrsubsampling[1];
                nrows = TIFFroundup(nrows, td->td_ycbcrsubsampling[1]);
                /* NB: don't need TIFFhowmany here 'cuz everything is rounded */
                return ((tsize_t)
                    (nrows*scanline + 2*(nrows*scanline / samplingarea)));
        } else
#endif
                return ((tsize_t)(nrows * TIFFScanlineSize(tif)));
}

/*
 * Compute the # bytes in a (row-aligned) strip.
 *
 * Note that if RowsPerStrip is larger than the
 * recorded ImageLength, then the strip size is
 * truncated to reflect the actual space required
 * to hold the strip.
 */
tsize_t
```

kfax'TIFFStripSize() (./kdegraphics/kfax/libtiffax/tif_strip.c:118)

```
TIFFStripSize(TIFF* tif)
{
        TIFFDirectory* td = &tif->tif_dir;
        uint32 rps = td->td_rowsperstrip;
        if (rps > td->td_imagelength)
                rps = td->td_imagelength;
        return (TIFFVStripSize(tif, rps));
}
```

```
/*
 * Compute a default strip size based on the image
 * characteristics and a requested value.  If the
 * request is <1 then we choose a strip size according
 * to certain heuristics.
 */
uint32
```

---

kfax'TIFFDefaultStripSize() (./kdegraphics/kfax/libtiffax/tif_strip.c:134)

```
TIFFDefaultStripSize(TIFF* tif, uint32 request)
{
        return (*tif->tif_defstripsize)(tif, request);
}

uint32
```

---

kfax'_TIFFDefaultStripSize() (./kdegraphics/kfax/libtiffax/tif_strip.c:140)

```
_TIFFDefaultStripSize(TIFF* tif, uint32 s)
{
        if ((int32) s < 1) {
                /*
                 * If RowsPerStrip is unspecified, try to break the
                 * image up into strips that are approximately 8Kbytes.
                 */
                tsize_t scanline = TIFFScanlineSize(tif);
                s = (uint32)(8*1024) / (scanline == 0 ? 1 : scanline);
                if (s == 0)             /* very wide images */
                        s = 1;
        }
        return (s);
}

/*
 * Return the number of bytes to read/write in a call to
 * one of the scanline-oriented i/o routines.  Note that
 * this number may be 1/samples-per-pixel if data is
 * stored as separate planes.
 */
tsize_t
```

---

kfax'TIFFScanlineSize() (./kdegraphics/kfax/libtiffax/tif_strip.c:162)

```
TIFFScanlineSize(TIFF* tif)
{
        TIFFDirectory *td = &tif->tif_dir;
        tsize_t scanline;

        scanline = td->td_bitspersample * td->td_imagewidth;
        if (td->td_planarconfig == PLANARCONFIG_CONTIG)
                scanline *= td->td_samplesperpixel;
        return ((tsize_t) TIFFhowmany(scanline, 8));
```

```
}

/*
 * Return the number of bytes required to store a complete
 * decoded and packed raster scanline (as opposed to the
 * I/O size returned by TIFFScanlineSize which may be less
 * if data is store as separate planes).
 */
tsize_t
```

---

kfax'TIFFRasterScanlineSize() (./kdegraphics/kfax/libtiffax/tif_strip.c:180)

```
TIFFRasterScanlineSize(TIFF* tif)
{
        TIFFDirectory *td = &tif->tif_dir;
        tsize_t scanline;

        scanline = td->td_bitspersample * td->td_imagewidth;
        if (td->td_planarconfig == PLANARCONFIG_CONTIG) {
                scanline *= td->td_samplesperpixel;
                return ((tsize_t) TIFFhowmany(scanline, 8));
        } else
                return ((tsize_t)
                    TIFFhowmany(scanline, 8)*td->td_samplesperpixel);
}
```

---

kfax'TIFFSwabShort() (./kdegraphics/kfax/libtiffax/tif_swab.c:36)

```
TIFFSwabShort(uint16* wp)
{
        register u_char* cp = (u_char*) wp;
        int t;

        t = cp[1]; cp[1] = cp[0]; cp[0] = t;
}
#endif

#ifndef TIFFSwabLong
void
```

---

kfax'TIFFSwabLong() (./kdegraphics/kfax/libtiffax/tif_swab.c:47)

```
TIFFSwabLong(uint32* lp)
{
        register u_char* cp = (u_char*) lp;
        int t;

        t = cp[3]; cp[3] = cp[0]; cp[0] = t;
        t = cp[2]; cp[2] = cp[1]; cp[1] = t;
}
#endif

#ifndef TIFFSwabArrayOfShort
```

```
void
```

---

kfax'TIFFSwabArrayOfShort() (./kdegraphics/kfax/libtiffax/tif_swab.c:59)

```
TIFFSwabArrayOfShort(uint16* wp, register u_long n)
{
        register u_char* cp;
        register int t;

        /* XXX unroll loop some */
        while (n-- > 0) {
                cp = (u_char*) wp;
                t = cp[1]; cp[1] = cp[0]; cp[0] = t;
                wp++;
        }
}
#endif

#ifndef TIFFSwabArrayOfLong
void
```

---

kfax'TIFFSwabArrayOfLong() (./kdegraphics/kfax/libtiffax/tif_swab.c:75)

```
TIFFSwabArrayOfLong(register uint32* lp, register u_long n)
{
        register unsigned char *cp;
        register int t;

        /* XXX unroll loop some */
        while (n-- > 0) {
                cp = (unsigned char *)lp;
                t = cp[3]; cp[3] = cp[0]; cp[0] = t;
                t = cp[2]; cp[2] = cp[1]; cp[1] = t;
                lp++;
        }
}
#endif

#ifndef TIFFSwabDouble
void
```

---

kfax'TIFFSwabDouble() (./kdegraphics/kfax/libtiffax/tif_swab.c:92)

```
TIFFSwabDouble(double *dp)
{
        register uint32* lp = (uint32*) dp;
        uint32 t;

        TIFFSwabArrayOfLong(lp, 2);
        t = lp[0]; lp[0] = lp[1]; lp[1] = t;
}
#endif
```

```
#ifndef TIFFSwabArrayOfDouble
void
```

---

kfax'TIFFSwabArrayOfDouble() (./kdegraphics/kfax/libtiffax/tif_swab.c:104)

```
TIFFSwabArrayOfDouble(double* dp, register u_long n)
{
        register uint32* lp = (uint32*) dp;
        register uint32 t;

        TIFFSwabArrayOfLong(lp, n + n);
        while (n-- > 0) {
                t = lp[0]; lp[0] = lp[1]; lp[1] = t;
                lp += 2;
        }
}
#endif

/*
 * Bit reversal tables.  TIFFBitRevTable[<byte>] gives
 * the bit reversed value of <byte>.  Used in various
 * places in the library when the FillOrder requires
 * bit reversal of byte values (e.g. CCITT Fax 3
 * encoding/decoding).  TIFFNoBitRevTable is provided
 * for algorithms that want an equivalent table that
 * do not reverse bit values.
 */
```

---

kfax'TIFFGetBitRevTable() (./kdegraphics/kfax/libtiffax/tif_swab.c:196)

```
TIFFGetBitRevTable(int reversed)
{
        return (reversed ? TIFFBitRevTable : TIFFNoBitRevTable);
}

void
```

---

kfax'TIFFReverseBits() (./kdegraphics/kfax/libtiffax/tif_swab.c:202)

```
TIFFReverseBits(register u_char* cp, register u_long n)
{
        for (; n > 8; n -= 8) {
                cp[0] = TIFFBitRevTable[cp[0]];
                cp[1] = TIFFBitRevTable[cp[1]];
                cp[2] = TIFFBitRevTable[cp[2]];
                cp[3] = TIFFBitRevTable[cp[3]];
                cp[4] = TIFFBitRevTable[cp[4]];
                cp[5] = TIFFBitRevTable[cp[5]];
                cp[6] = TIFFBitRevTable[cp[6]];
                cp[7] = TIFFBitRevTable[cp[7]];
                cp += 8;
        }
        while (n-- > 0)
```

```
                *cp = TIFFBitRevTable[*cp], cp++;
}
```

---

kfax'ThunderDecode() (./kdegraphics/kfax/libtiffax/tif_thunder.c:67)

```
ThunderDecode(TIFF* tif, tidata_t op, tsize_t maxpixels)
{
        register u_char *bp;
        register tsize_t cc;
        u_int lastpixel;
        tsize_t npixels;

        bp = (u_char *)tif->tif_rawcp;
        cc = tif->tif_rawcc;
        lastpixel = 0;
        npixels = 0;
        while (cc > 0 && npixels < maxpixels) {
                int n, delta;

                n = *bp++, cc--;
                switch (n & THUNDER_CODE) {
                case THUNDER_RUN:               /* pixel run */
                        /*
                         * Replicate the last pixel n times,
                         * where n is the lower-order 6 bits.
                         */
                        if (npixels & 1) {
                                op[0] |= lastpixel;
                                lastpixel = *op++; npixels++; n--;
                        } else
                                lastpixel |= lastpixel << 4;
                        npixels += n;
                        for (; n > 0; n -= 2)
                                *op++ = lastpixel;
                        if (n == -1)
                                *--op &= 0xf0;
                        lastpixel &= 0xf;
                        break;
                case THUNDER_2BITDELTAS:        /* 2-bit deltas */
                        if ((delta = ((n >> 4) & 3)) != DELTA2_SKIP)
                                SETPIXEL(op, lastpixel + twobitdeltas[delta]);
                        if ((delta = ((n >> 2) & 3)) != DELTA2_SKIP)
                                SETPIXEL(op, lastpixel + twobitdeltas[delta]);
                        if ((delta = (n & 3)) != DELTA2_SKIP)
                                SETPIXEL(op, lastpixel + twobitdeltas[delta]);
                        break;
                case THUNDER_3BITDELTAS:        /* 3-bit deltas */
                        if ((delta = ((n >> 3) & 7)) != DELTA3_SKIP)
                                SETPIXEL(op, lastpixel + threebitdeltas[delta]);
                        if ((delta = (n & 7)) != DELTA3_SKIP)
                                SETPIXEL(op, lastpixel + threebitdeltas[delta]);
                        break;
                case THUNDER_RAW:               /* raw data */
                        SETPIXEL(op, n);
                        break;
                }
        }
        tif->tif_rawcp = (tidata_t) bp;
```

```
        tif->tif_rawcc = cc;
        if (npixels != maxpixels) {
                TIFFError(tif->tif_name,
                    "ThunderDecode: %s data at scanline %ld (%lu != %lu)",
                    npixels < maxpixels ? "Not enough" : "Too much",
                    (long) tif->tif_row, (long) npixels, (long) maxpixels);
                return (0);
        }
        return (1);
}

static int
```

---

kfax'ThunderDecodeRow() (./kdegraphics/kfax/libtiffax/tif_thunder.c:132)

```
ThunderDecodeRow(TIFF* tif, tidata_t buf, tsize_t occ, tsample_t s)
{
        tidata_t row = buf;

        (void) s;
        while ((long)occ > 0) {
                if (!ThunderDecode(tif, row, tif->tif_dir.td_imagewidth))
                        return (0);
                occ -= tif->tif_scanlinesize;
                row += tif->tif_scanlinesize;
        }
        return (1);
}

int
```

---

kfax'TIFFInitThunderScan() (./kdegraphics/kfax/libtiffax/tif_thunder.c:147)

```
TIFFInitThunderScan(TIFF* tif, int scheme)
{
        (void) scheme;
        tif->tif_decoderow = ThunderDecodeRow;
        tif->tif_decodestrip = ThunderDecodeRow;
        return (1);
}
```

---

kfax'TIFFComputeTile() (./kdegraphics/kfax/libtiffax/tif_tile.c:38)

```
TIFFComputeTile(TIFF* tif, uint32 x, uint32 y, uint32 z, tsample_t s)
{
        TIFFDirectory *td = &tif->tif_dir;
        uint32 dx = td->td_tilewidth;
        uint32 dy = td->td_tilelength;
        uint32 dz = td->td_tiledepth;
        ttile_t tile = 1;

        if (td->td_imagedepth == 1)
                z = 0;
```

```
        if (dx == (uint32) -1)
                dx = td->td_imagewidth;
        if (dy == (uint32) -1)
                dy = td->td_imagelength;
        if (dz == (uint32) -1)
                dz = td->td_imagedepth;
        if (dx != 0 && dy != 0 && dz != 0) {
                uint32 xpt = TIFFhowmany(td->td_imagewidth, dx);
                uint32 ypt = TIFFhowmany(td->td_imagelength, dy);
                uint32 zpt = TIFFhowmany(td->td_imagedepth, dz);

                if (td->td_planarconfig == PLANARCONFIG_SEPARATE)
                        tile = (xpt*ypt*zpt)*s +
                             (xpt*ypt)*(z/dz) +
                             xpt*(y/dy) +
                             x/dx;
                else
                        tile = (xpt*ypt)*(z/dz) + xpt*(y/dy) + x/dx + s;
        }
        return (tile);
}

/*
 * Check an (x,y,z,s) coordinate
 * against the image bounds.
 */
int
```

---

kfax'TIFFCheckTile() (./kdegraphics/kfax/libtiffax/tif_tile.c:75)

```
TIFFCheckTile(TIFF* tif, uint32 x, uint32 y, uint32 z, tsample_t s)
{
        TIFFDirectory *td = &tif->tif_dir;

        if (x >= td->td_imagewidth) {
                TIFFError(tif->tif_name, "Col %ld out of range, max %lu",
                    (long) x, (u_long) td->td_imagewidth);
                return (0);
        }
        if (y >= td->td_imagelength) {
                TIFFError(tif->tif_name, "Row %ld out of range, max %lu",
                    (long) y, (u_long) td->td_imagelength);
                return (0);
        }
        if (z >= td->td_imagedepth) {
                TIFFError(tif->tif_name, "Depth %ld out of range, max %lu",
                    (long) z, (u_long) td->td_imagedepth);
                return (0);
        }
        if (td->td_planarconfig == PLANARCONFIG_SEPARATE &&
            s >= td->td_samplesperpixel) {
                TIFFError(tif->tif_name, "Sample %d out of range, max %u",
                    (int) s, td->td_samplesperpixel);
                return (0);
        }
        return (1);
}
```

```
/*
 * Compute how many tiles are in an image.
 */
ttile_t
```

---

kfax'TIFFNumberOfTiles() (./kdegraphics/kfax/libtiffax/tif_tile.c:107)

```
TIFFNumberOfTiles(TIFF* tif)
{
        TIFFDirectory *td = &tif->tif_dir;
        uint32 dx = td->td_tilewidth;
        uint32 dy = td->td_tilelength;
        uint32 dz = td->td_tiledepth;
        ttile_t ntiles;

        if (dx == (uint32) -1)
                dx = td->td_imagewidth;
        if (dy == (uint32) -1)
                dy = td->td_imagelength;
        if (dz == (uint32) -1)
                dz = td->td_imagedepth;
        ntiles = (dx == 0 || dy == 0 || dz == 0) ? 0 :
            (TIFFhowmany(td->td_imagewidth, dx) *
             TIFFhowmany(td->td_imagelength, dy) *
             TIFFhowmany(td->td_imagedepth, dz));
        if (td->td_planarconfig == PLANARCONFIG_SEPARATE)
                ntiles *= td->td_samplesperpixel;
        return (ntiles);
}

/*
 * Compute the # bytes in each row of a tile.
 */
tsize_t
```

---

kfax'TIFFTileRowSize() (./kdegraphics/kfax/libtiffax/tif_tile.c:134)

```
TIFFTileRowSize(TIFF* tif)
{
        TIFFDirectory *td = &tif->tif_dir;
        tsize_t rowsize;

        if (td->td_tilelength == 0 || td->td_tilewidth == 0)
                return ((tsize_t) 0);
        rowsize = td->td_bitspersample * td->td_tilewidth;
        if (td->td_planarconfig == PLANARCONFIG_CONTIG)
                rowsize *= td->td_samplesperpixel;
        return ((tsize_t) TIFFhowmany(rowsize, 8));
}

/*
 * Compute the # bytes in a variable length, row-aligned tile.
 */
tsize_t
```

---

kfax'TIFFVTileSize() (./kdegraphics/kfax/libtiffax/tif_tile.c:151)

```
TIFFVTileSize(TIFF* tif, uint32 nrows)
{
        TIFFDirectory *td = &tif->tif_dir;
        tsize_t tilesize;

        if (td->td_tilelength == 0 || td->td_tilewidth == 0 ||
            td->td_tiledepth == 0)
                return ((tsize_t) 0);
#ifdef YCBCR_SUPPORT
        if (td->td_planarconfig == PLANARCONFIG_CONTIG &&
            td->td_photometric == PHOTOMETRIC_YCBCR &&
            !isUpSampled(tif)) {
                /*
                 * Packed YCbCr data contain one Cb+Cr for every
                 * HorizontalSampling*VerticalSampling Y values.
                 * Must also roundup width and height when calculating
                 * since images that are not a multiple of the
                 * horizontal/vertical subsampling area include
                 * YCbCr data for the extended image.
                 */
                tsize_t w =
                    TIFFroundup(td->td_tilewidth, td->td_ycbcrsubsampling[0]);
                tsize_t rowsize = TIFFhowmany(w*td->td_bitspersample, 8);
                tsize_t samplingarea =
                    td->td_ycbcrsubsampling[0]*td->td_ycbcrsubsampling[1];
                nrows = TIFFroundup(nrows, td->td_ycbcrsubsampling[1]);
                /* NB: don't need TIFFhowmany here 'cuz everything is rounded */
                tilesize = nrows*rowsize + 2*(nrows*rowsize / samplingarea);
        } else
#endif
                tilesize = nrows * TIFFTileRowSize(tif);
        return ((tsize_t)(tilesize * td->td_tiledepth));
}

/*
 * Compute the # bytes in a row-aligned tile.
 */
tsize_t
```

---

kfax'TIFFTileSize() (./kdegraphics/kfax/libtiffax/tif_tile.c:189)

```
TIFFTileSize(TIFF* tif)
{
        return (TIFFVTileSize(tif, tif->tif_dir.td_tilelength));
}

/*
 * Compute a default tile size based on the image
 * characteristics and a requested value.  If a
 * request is <1 then we choose a size according
 * to certain heuristics.
 */
void
```

---

kfax'TIFFDefaultTileSize() (./kdegraphics/kfax/libtiffax/tif_tile.c:201)

```
TIFFDefaultTileSize(TIFF* tif, uint32* tw, uint32* th)
{
        (*tif->tif_deftilesize)(tif, tw, th);
}

void
```

---

kfax'_TIFFDefaultTileSize() (./kdegraphics/kfax/libtiffax/tif_tile.c:207)

```
_TIFFDefaultTileSize(TIFF* tif, uint32* tw, uint32* th)
{
        (void) tif;
        if (*(int32*) tw < 1)
                *tw = 256;
        if (*(int32*) th < 1)
                *th = 256;
        /* roundup to a multiple of 16 per the spec */
        if (*tw & 0xf)
                *tw = TIFFroundup(*tw, 16);
        if (*th & 0xf)
                *th = TIFFroundup(*th, 16);
}
```

---

kfax'_tiffReadProc() (./kdegraphics/kfax/libtiffax/tif_unix.c:36)

```
_tiffReadProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        return ((tsize_t) read((int) fd, buf, (size_t) size));
}

static tsize_t
```

---

kfax'_tiffWriteProc() (./kdegraphics/kfax/libtiffax/tif_unix.c:42)

```
_tiffWriteProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        return ((tsize_t) write((int) fd, buf, (size_t) size));
}

static toff_t
```

---

kfax'_tiffSeekProc() (./kdegraphics/kfax/libtiffax/tif_unix.c:48)

```
_tiffSeekProc(thandle_t fd, toff_t off, int whence)
{
        return ((toff_t) lseek((int) fd, (off_t) off, whence));
}
```

```
static int
```

---

kfax'_tiffCloseProc() (./kdegraphics/kfax/libtiffax/tif_unix.c:54)

```
_tiffCloseProc(thandle_t fd)
{
        return (close((int) fd));
}

#include <sys/stat.h>

static toff_t
```

---

kfax'_tiffSizeProc() (./kdegraphics/kfax/libtiffax/tif_unix.c:62)

```
_tiffSizeProc(thandle_t fd)
{
#ifdef _AM29K
        long fsize;
        return ((fsize = lseek((int) fd, 0, SEEK_END)) < 0 ? 0 : fsize);
#else
        struct stat sb;
        return (toff_t) (fstat((int) fd, &sb) < 0 ? 0 : sb.st_size);
#endif
}

#ifdef HAVE_MMAP
#include <sys/mman.h>

static int
```

---

kfax'_tiffMapProc() (./kdegraphics/kfax/libtiffax/tif_unix.c:77)

```
_tiffMapProc(thandle_t fd, tdata_t* pbase, toff_t* psize)
{
        toff_t size = _tiffSizeProc(fd);
        if (size != (toff_t) -1) {
                *pbase = (tdata_t)
                    mmap(0, size, PROT_READ, MAP_SHARED, (int) fd, 0);
                if (*pbase != (tdata_t) -1) {
                        *psize = size;
                        return (1);
                }
        }
        return (0);
}

static void
```

---

kfax'_tiffUnmapProc() (./kdegraphics/kfax/libtiffax/tif_unix.c:92)

```
_tiffUnmapProc(thandle_t fd, tdata_t base, toff_t size)
{
        (void) fd;
        (void) munmap(base, (off_t) size);
}
#else /* !HAVE_MMAP */
static int
```

---

kfax'_tiffMapProc() (./kdegraphics/kfax/libtiffax/tif_unix.c:99)

```
_tiffMapProc(thandle_t fd, tdata_t* pbase, toff_t* psize)
{
        (void) fd; (void) pbase; (void) psize;
        return (0);
}

static void
```

---

kfax'_tiffUnmapProc() (./kdegraphics/kfax/libtiffax/tif_unix.c:106)

```
_tiffUnmapProc(thandle_t fd, tdata_t base, toff_t size)
{
        (void) fd; (void) base; (void) size;
}
#endif /* !HAVE_MMAP */

/*
 * Open a TIFF file descriptor for read/writing.
 */
TIFF*
```

---

kfax'TIFFFdOpen() (./kdegraphics/kfax/libtiffax/tif_unix.c:116)

```
TIFFFdOpen(int fd, const char* name, const char* mode)
{
        TIFF* tif;

        tif = TIFFClientOpen(name, mode,
            (thandle_t) fd,
            _tiffReadProc, _tiffWriteProc,
            _tiffSeekProc, _tiffCloseProc, _tiffSizeProc,
            _tiffMapProc, _tiffUnmapProc);
        if (tif)
                tif->tif_fd = fd;
        return (tif);
}

/*
 * Open a TIFF file for read/writing.
 */
TIFF*
```

kfax'TIFFOpen() (./kdegraphics/kfax/libtiffax/tif_unix.c:134)

```
TIFFOpen(const char* name, const char* mode)
{
        static const char module[] = "TIFFOpen";
        int m, fd;

        m = _TIFFgetMode(mode, module);
        if (m == -1)
                return ((TIFF*)0);
#ifdef _AM29K
        fd = open(name, m);
#else
        fd = open(name, m, 0666);
#endif
        if (fd < 0) {
                TIFFError(module, "%s: Cannot open", name);
                return ((TIFF *)0);
        }
        return (TIFFFdOpen(fd, name, mode));
}

void*
```

kfax'_TIFFmalloc() (./kdegraphics/kfax/libtiffax/tif_unix.c:155)

```
_TIFFmalloc(tsize_t s)
{
        return (malloc((size_t) s));
}

void
```

kfax'_TIFFfree() (./kdegraphics/kfax/libtiffax/tif_unix.c:161)

```
_TIFFfree(tdata_t p)
{
        free(p);
}

void*
```

kfax'_TIFFrealloc() (./kdegraphics/kfax/libtiffax/tif_unix.c:167)

```
_TIFFrealloc(tdata_t p, tsize_t s)
{
        return (realloc(p, (size_t) s));
}

void
```

---

kfax'_TIFFmemset() (./kdegraphics/kfax/libtiffax/tif_unix.c:173)

```
_TIFFmemset(tdata_t p, int v, tsize_t c)
{
        memset(p, v, (size_t) c);
}

void
```

---

kfax'_TIFFmemcpy() (./kdegraphics/kfax/libtiffax/tif_unix.c:179)

```
_TIFFmemcpy(tdata_t d, const tdata_t s, tsize_t c)
{
        memcpy(d, s, (size_t) c);
}

int
```

---

kfax'_TIFFmemcmp() (./kdegraphics/kfax/libtiffax/tif_unix.c:185)

```
_TIFFmemcmp(const tdata_t p1, const tdata_t p2, tsize_t c)
{
        return (memcmp(p1, p2, (size_t) c));
}

static void
```

---

kfax'unixWarningHandler() (./kdegraphics/kfax/libtiffax/tif_unix.c:191)

```
unixWarningHandler(const char* module, const char* fmt, va_list ap)
{
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        fprintf(stderr, "Warning, ");
        vfprintf(stderr, fmt, ap);
        fprintf(stderr, ".\n");
}
```

---

kfax'unixErrorHandler() (./kdegraphics/kfax/libtiffax/tif_unix.c:202)

```
unixErrorHandler(const char* module, const char* fmt, va_list ap)
{
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        vfprintf(stderr, fmt, ap);
        fprintf(stderr, ".\n");
}
```

---

kfax'TIFFGetVersion() (./kdegraphics/kfax/libtiffax/tif_version.c:31)

```
TIFFGetVersion(void)
{
        return (TIFFVersion);
}
```

---

kfax'TIFFModeCCITTFax3() (./kdegraphics/kfax/libtiffax/tif_vms.c:46)

```
void TIFFModeCCITTFax3(void){}
#endif

static tsize_t
```

---

kfax'_tiffReadProc() (./kdegraphics/kfax/libtiffax/tif_vms.c:50)

```
_tiffReadProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        return (read((int) fd, buf, size));
}

static tsize_t
```

---

kfax'_tiffWriteProc() (./kdegraphics/kfax/libtiffax/tif_vms.c:56)

```
_tiffWriteProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        return (write((int) fd, buf, size));
}

static toff_t
```

---

kfax'_tiffSeekProc() (./kdegraphics/kfax/libtiffax/tif_vms.c:62)

```
_tiffSeekProc(thandle_t fd, toff_t off, int whence)
{
        return ((toff_t) lseek((int) fd, (off_t) off, whence));
}

static int
```

---

kfax'_tiffCloseProc() (./kdegraphics/kfax/libtiffax/tif_vms.c:68)

```
_tiffCloseProc(thandle_t fd)
{
```

```
        return (close((int) fd));
}

#include <sys/stat.h>

static toff_t
```

---

kfax'_tiffSizeProc() (./kdegraphics/kfax/libtiffax/tif_vms.c:76)

```
_tiffSizeProc(thandle_t fd)
{
        struct stat sb;
        return (toff_t) (fstat((int) fd, &sb) < 0 ? 0 : sb.st_size);
}

#ifdef HAVE_MMAP
#include <starlet.h>
#include <fab.h>
#include <secdef.h>

/*
 * Table for storing information on current open sections.
 * (Should really be a linked list)
 */
```

---

kfax'_tiffMapProc() (./kdegraphics/kfax/libtiffax/tif_vms.c:113)

```
_tiffMapProc(thandle_t fd, tdata_t* pbase, toff_t* psize)
{
        char name[256];
        struct FAB fab;
        unsigned short channel;
        char *inadr[2], *retadr[2];
        unsigned long status;
        long size;

        if (no_mapped >= MAX_MAPPED)
                return(0);
        /*
         * We cannot use a file descriptor, we
         * must open the file once more.
         */
        if (getname((int)fd, name, 1) == NULL)
                return(0);
        /* prepare the FAB for a user file open */
        fab = cc$rms_fab;
        fab.fab$l_fop |= FAB$V_UFO;
        fab.fab$b_fac = FAB$M_GET;
        fab.fab$b_shr = FAB$M_SHRGET;
        fab.fab$l_fna = name;
        fab.fab$b_fns = strlen(name);
        status = sys$open(&fab);        /* open file & get channel number */
        if ((status&1) == 0)
                return(0);
        channel = (unsigned short)fab.fab$l_stv;
        inadr[0] = inadr[1] = (char *)0; /* just an address in P0 space */
```

```
        /*
         * Map the blocks of the file up to
         * the EOF block into virtual memory.
         */
        size = _tiffSizeProc(fd);
        status = sys$crmpsc(inadr, retadr, 0, SEC$M_EXPREG, 0,0,0, channel,
                TIFFhowmany(size,512), 0,0,0);
        if ((status&1) == 0){
                sys$dassgn(channel);
                return(0);
        }
        *pbase = (tdata_t) retadr[0];   /* starting virtual address */
        /*
         * Use the size of the file up to the
         * EOF mark for UNIX compatibility.
         */
        *psize = (toff_t) size;
        /* Record the section in the table */
        map_table[no_mapped].base = retadr[0];
        map_table[no_mapped].top = retadr[1];
        map_table[no_mapped].channel = channel;
        no_mapped++;

        return(1);
}

/*
 * This routine unmaps a section from the virtual address space of
 * the process, but only if the base was the one returned from a
 * call to TIFFMapFileContents.
 */
static void
```

---

kfax'_tiffUnmapProc() (./kdegraphics/kfax/libtiffax/tif_vms.c:174)

```
_tiffUnmapProc(thandle_t fd, tdata_t base, toff_t size)
{
        char *inadr[2];
        int i, j;

        /* Find the section in the table */
        for (i = 0;i < no_mapped; i++) {
                if (map_table[i].base == (char *) base) {
                        /* Unmap the section */
                        inadr[0] = (char *) base;
                        inadr[1] = map_table[i].top;
                        sys$deltva(inadr, 0, 0);
                        sys$dassgn(map_table[i].channel);
                        /* Remove this section from the list */
                        for (j = i+1; j < no_mapped; j++)
                                map_table[j-1] = map_table[j];
                        no_mapped--;
                        return;
                }
        }
}
#else /* !HAVE_MMAP */
static int
```

---

kfax'_tiffMapProc() (./kdegraphics/kfax/libtiffax/tif_vms.c:197)

```
_tiffMapProc(thandle_t fd, tdata_t* pbase, toff_t* psize)
{
        return (0);
}

static void
```

---

kfax'_tiffUnmapProc() (./kdegraphics/kfax/libtiffax/tif_vms.c:203)

```
_tiffUnmapProc(thandle_t fd, tdata_t base, toff_t size)
{
}
#endif /* !HAVE_MMAP */

/*
 * Open a TIFF file descriptor for read/writing.
 */
TIFF*
```

---

kfax'TIFFFdOpen() (./kdegraphics/kfax/libtiffax/tif_vms.c:212)

```
TIFFFdOpen(int fd, const char* name, const char* mode)
{
        TIFF* tif;

        tif = TIFFClientOpen(name, mode,
            (thandle_t) fd,
            _tiffReadProc, _tiffWriteProc, _tiffSeekProc, _tiffCloseProc,
            _tiffSizeProc, _tiffMapProc, _tiffUnmapProc);
        if (tif)
                tif->tif_fd = fd;
        return (tif);
}

/*
 * Open a TIFF file for read/writing.
 */
TIFF*
```

---

kfax'TIFFOpen() (./kdegraphics/kfax/libtiffax/tif_vms.c:229)

```
TIFFOpen(const char* name, const char* mode)
{
        static const char module[] = "TIFFOpen";
        int m, fd;

        m = _TIFFgetMode(mode, module);
        if (m == -1)
```

```
                return ((TIFF*)0);
        if (m&O_TRUNC){
                /*
                 * There is a bug in open in VAXC. If you use
                 * open w/ m=O_RDWR|O_CREAT|O_TRUNC the
                 * wrong thing happens.  On the other hand
                 * creat does the right thing.
                 */
                fd = creat((char *) /* bug in stdio.h */ name, 0666,
                    "alq = 128", "deq = 64", "mbc = 32",
                    "fop = tef");
        } else if (m&O_RDWR) {
                fd = open(name, m, 0666,
                    "deq = 64", "mbc = 32", "fop = tef", "ctx = stm");
        } else
                fd = open(name, m, 0666, "mbc = 32", "ctx = stm");
        if (fd < 0) {
                TIFFError(module, "%s: Cannot open", name);
                return ((TIFF*)0);
        }
        return (TIFFFdOpen(fd, name, mode));
}

tdata_t
```

---

kfax'_TIFFmalloc() (./kdegraphics/kfax/libtiffax/tif_vms.c:260)

```
_TIFFmalloc(tsize_t s)
{
        return (malloc((size_t) s));
}

void
```

---

kfax'_TIFFfree() (./kdegraphics/kfax/libtiffax/tif_vms.c:266)

```
_TIFFfree(tdata_t p)
{
        free(p);
}

tdata_t
```

---

kfax'_TIFFrealloc() (./kdegraphics/kfax/libtiffax/tif_vms.c:272)

```
_TIFFrealloc(tdata_t p, tsize_t s)
{
        return (realloc(p, (size_t) s));
}

void
```

---

kfax'_TIFFmemset() (./kdegraphics/kfax/libtiffax/tif_vms.c:278)

```
_TIFFmemset(tdata_t p, int v, tsize_t c)
{
        memset(p, v, (size_t) c);
}

void
```

---

kfax'_TIFFmemcpy() (./kdegraphics/kfax/libtiffax/tif_vms.c:284)

```
_TIFFmemcpy(tdata_t d, const void* s, tsize_t c)
{
        memcpy(d, s, (size_t) c);
}

int
```

---

kfax'_TIFFmemcmp() (./kdegraphics/kfax/libtiffax/tif_vms.c:290)

```
_TIFFmemcmp(const tdata_t p1, const tdata_t p2, tsize_t c)
{
        return (memcmp(p1, p2, (size_t) c));
}

/*
 * On the VAX, we need to make those global, writable pointers
 * non-shareable, otherwise they would be made shareable by default.
 * On the AXP, this brain damage has been corrected.
 *
 * I (Karsten Spang, krs@kampsax.dk) have dug around in the GCC
 * manual and the GAS code and have come up with the following
 * construct, but I don't have GCC on my VAX, so it is untested.
 * Please tell me if it does not work.
 */

static void
```

---

kfax'vmsWarningHandler() (./kdegraphics/kfax/libtiffax/tif_vms.c:307)

```
vmsWarningHandler(const char* module, const char* fmt, va_list ap)
{
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        fprintf(stderr, "Warning, ");
        vfprintf(stderr, fmt, ap);
        fprintf(stderr, ".\n");
}
```

---

kfax'vmsErrorHandler() (./kdegraphics/kfax/libtiffax/tif_vms.c:323)

```
vmsErrorHandler(const char* module, const char* fmt, va_list ap)
{
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        vfprintf(stderr, fmt, ap);
        fprintf(stderr, ".\n");
}
```

---

kfax'ieeetod() (./kdegraphics/kfax/libtiffax/tif_vms.c:415)

```
ieeetod(double *dp)
{
        double_t source;
        long sign,exp,mant;
        double dmant;

        source.d = *dp;
        sign = source.ieee.sign;
        exp = source.ieee.exp;
        mant = source.ieee.mant;

        if (exp == 2047) {
                if (mant)                       /* Not a Number (NAN) */
                        *dp = HUGE_VAL;
                else                            /* +/- infinity */
                        *dp = (sign ? -HUGE_VAL : HUGE_VAL);
                return;
        }
        if (!exp) {
                if (!(mant || source.ieee.mant2)) {     /* zero */
                        *dp=0;
                        return;
                } else {                        /* Unnormalized number */
                        /* NB: not -1023, the 1 bit is not implied */
                        exp= -1022;
                }
        } else {
                mant |= 1<<20;
                exp -= 1023;
        }
        dmant = (((double) mant) +
                ((double) source.ieee.mant2) / (((double) (1<<16)) *
                ((double) (1<<16)))) / (double) (1<<20);
        dmant = ldexp(dmant, exp);
        if (sign)
                dmant= -dmant;
        *dp = dmant;
}

INLINE static void
```

---

kfax'dtoieee() (./kdegraphics/kfax/libtiffax/tif_vms.c:455)

```
dtoieee(double *dp)
{
        double_t num;
        double x;
        int exp;

        num.d = *dp;
        if (!num.d) {                   /* Zero is just binary all zeros */
                num.l[0] = num.l[1] = 0;
                return;
        }

        if (num.d < 0) {                /* Sign is encoded separately */
                num.d = -num.d;
                num.ieee.sign = 1;
        } else {
                num.ieee.sign = 0;
        }

        /* Now separate the absolute value into mantissa and exponent */
        x = frexp(num.d, &exp);

        /*
         * Handle cases where the value is outside the
         * range for IEEE floating point numbers.
         * (Overflow cannot happen on a VAX, but underflow
         * can happen for G float.)
         */
        if (exp < -1022) {              /* Unnormalized number */
                x = ldexp(x, -1023-exp);
                exp = 0;
        } else if (exp > 1023) {        /* +/- infinity */
                x = 0;
                exp = 2047;
        } else {                        /* Get rid of most significant bit */
                x *= 2;
                x -= 1;
                exp += 1023;
        }
        num.ieee.exp = exp;

        x *= (double) (1<<20);
        num.ieee.mant = (long) x;
        x -= (double) num.ieee.mant;
        num.ieee.mant2 = (long) (x*((double) (1<<16)*(double) (1<<16)));

        if (!(num.ieee.mant || num.ieee.exp || num.ieee.mant2)) {
                /* Avoid negative zero */
                num.ieee.sign = 0;
        }
        *dp = num.d;
}

/*
 * Beware, these do not handle over/under-flow
 * during conversion from ieee to native format.
 */
```

---

kfax'TIFFCvtIEEEFloatToNative() (./kdegraphics/kfax/libtiffax/tif_vms.c:542)

```
TIFFCvtIEEEFloatToNative(TIFF* tif, u_int n, float* f)
{
        float_t* fp = (float_t*) f;

        while (n-- > 0) {
                IEEEFLOAT2NATIVE(fp);
                fp++;
        }
}

void
```

---

kfax'TIFFCvtNativeToIEEEFloat() (./kdegraphics/kfax/libtiffax/tif_vms.c:553)

```
TIFFCvtNativeToIEEEFloat(TIFF* tif, u_int n, float* f)
{
        float_t* fp = (float_t*) f;

        while (n-- > 0) {
                NATIVE2IEEEFLOAT(fp);
                fp++;
        }
}
void
```

---

kfax'TIFFCvtIEEEDoubleToNative() (./kdegraphics/kfax/libtiffax/tif_vms.c:563)

```
TIFFCvtIEEEDoubleToNative(TIFF* tif, u_int n, float* f)
{
        float_t* fp = (float_t*) f;

        while (n-- > 0) {
                IEEEDOUBLE2NATIVE(fp);
                fp++;
        }
}

void
```

---

kfax'TIFFCvtNativeToIEEEDouble() (./kdegraphics/kfax/libtiffax/tif_vms.c:574)

```
TIFFCvtNativeToIEEEDouble(TIFF* tif, u_int n, float* f)
{
        float_t* fp = (float_t*) f;
```

```
        while (n-- > 0) {
                NATIVE2IEEEDOUBLE(fp);
                fp++;
        }
}
```

---

kfax'TIFFSetWarningHandler() (./kdegraphics/kfax/libtiffax/tif_warning.c:33)

```
TIFFSetWarningHandler(TIFFErrorHandler handler)
{
        TIFFErrorHandler prev = _TIFFwarningHandler;
        _TIFFwarningHandler = handler;
        return (prev);
}

void
```

---

kfax'TIFFWarning() (./kdegraphics/kfax/libtiffax/tif_warning.c:41)

```
TIFFWarning(const char* module, const char* fmt, ...)
{
        if (_TIFFwarningHandler) {
                va_list ap;
                va_start(ap, fmt);
                (*_TIFFwarningHandler)(module, fmt, ap);
                va_end(ap);
        }
}
```

---

kfax'_tiffReadProc() (./kdegraphics/kfax/libtiffax/tif_win3.c:40)

```
_tiffReadProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        return (_hread(fd, buf, size));
}

static tsize_t
```

---

kfax'_tiffWriteProc() (./kdegraphics/kfax/libtiffax/tif_win3.c:46)

```
_tiffWriteProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        return (_hwrite(fd, buf, size));
}

static toff_t
```

---

kfax'_tiffSeekProc() (./kdegraphics/kfax/libtiffax/tif_win3.c:52)

```
_tiffSeekProc(thandle_t fd, toff_t off, int whence)
{
        return (_llseek(fd, (off_t) off, whence));
}

static int
```

---

kfax'_tiffCloseProc() (./kdegraphics/kfax/libtiffax/tif_win3.c:58)

```
_tiffCloseProc(thandle_t fd)
{
        return (_lclose(fd));
}

#include <sys/stat.h>

static toff_t
```

---

kfax'_tiffSizeProc() (./kdegraphics/kfax/libtiffax/tif_win3.c:66)

```
_tiffSizeProc(thandle_t fd)
{
        struct stat sb;
        return (fstat((int) fd, &sb) < 0 ? 0 : sb.st_size);
}

static int
```

---

kfax'_tiffMapProc() (./kdegraphics/kfax/libtiffax/tif_win3.c:73)

```
_tiffMapProc(thandle_t fd, tdata_t* pbase, toff_t* psize)
{
        return (0);
}

static void
```

---

kfax'_tiffUnmapProc() (./kdegraphics/kfax/libtiffax/tif_win3.c:79)

```
_tiffUnmapProc(thandle_t fd, tdata_t base, toff_t size)
{
}

/*
 * Open a TIFF file descriptor for read/writing.
 */
TIFF*
```

---

kfax'TIFFFdOpen() (./kdegraphics/kfax/libtiffax/tif_win3.c:87)

```
TIFFFdOpen(int fd, const char* name, const char* mode)
{
        TIFF* tif;

        tif = TIFFClientOpen(name, mode,
            (void*) fd,
            _tiffReadProc, _tiffWriteProc, _tiffSeekProc, _tiffCloseProc,
            _tiffSizeProc, _tiffMapProc, _tiffUnmapProc);
        if (tif)
                tif->tif_fd = fd;
        return (tif);
}

/*
 * Open a TIFF file for read/writing.
 */
TIFF*
```

---

kfax'TIFFOpen() (./kdegraphics/kfax/libtiffax/tif_win3.c:104)

```
TIFFOpen(const char* name, const char* mode)
{
        static const char module[] = "TIFFOpen";
        int m, fd;
        OFSTRUCT of;
        int mm = 0;

        m = _TIFFgetMode(mode, module);
        if (m == -1)
                return ((TIFF*)0);
        if (m & O_CREAT) {
                if ((m & O_TRUNC) || OpenFile(name, &of, OF_EXIST) != HFILE_ERRO
                        mm |= OF_CREATE;
        }
        if (m & O_WRONLY)
                mm |= OF_WRITE;
        if (m & O_RDWR)
                mm |= OF_READWRITE;
        fd = OpenFile(name, &of, mm);
        if (fd < 0) {
                TIFFError(module, "%s: Cannot open", name);
                return ((TIFF*)0);
        }
        return (TIFFFdOpen(fd, name, mode));
}

tdata_t
```

---

kfax'_TIFFmalloc() (./kdegraphics/kfax/libtiffax/tif_win3.c:131)

```
_TIFFmalloc(tsize_t s)
{
```

```
        return (tdata_t) GlobalAllocPtr(GHND, (DWORD) s);
}

void
```

---

kfax'_TIFFfree() (./kdegraphics/kfax/libtiffax/tif_win3.c:137)

```
_TIFFfree(tdata_t p)
{
        GlobalFreePtr(p);
}

tdata_t
```

---

kfax'_TIFFrealloc() (./kdegraphics/kfax/libtiffax/tif_win3.c:143)

```
_TIFFrealloc(tdata_t p, tsize_t s)
{
        return (tdata_t) GlobalReAllocPtr(p, (DWORD) s, GHND);
}

void
```

---

kfax'_TIFFmemset() (./kdegraphics/kfax/libtiffax/tif_win3.c:149)

```
_TIFFmemset(tdata_t p, int v, tsize_t c)
{
        char* pp = (char*) p;

        while (c > 0) {
                tsize_t chunk = 0x10000 - ((uint32) pp & 0xffff);/* What's left 
                if (chunk > 0xff00)                             /* No more than 
                        chunk = 0xff00;
                if (chunk > c)                                  /* No more than 
                        chunk = c;
                memset(pp, v, chunk);
                pp = (char*) (chunk + (char huge*) pp);
                c -= chunk;
        }
}

void
```

---

kfax'_TIFFmemcpy() (./kdegraphics/kfax/libtiffax/tif_win3.c:166)

```
_TIFFmemcpy(tdata_t d, const tdata_t s, tsize_t c)
{
        if (c > 0xFFFF)
                hmemcpy((void _huge*) d, (void _huge*) s, c);
        else
                (void) memcpy(d, s, (size_t) c);
```

```
}

int
```

kfax'_TIFFmemcmp() (./kdegraphics/kfax/libtiffax/tif_win3.c:175)

```
_TIFFmemcmp(const tdata_t d, const tdata_t s, tsize_t c)
{
        char* dd = (char*) d;
        char* ss = (char*) s;
        tsize_t chunks, chunkd, chunk;
        int result;

        while (c > 0) {
                chunks = 0x10000 - ((uint32) ss & 0xffff);      /* What's left i
                chunkd = 0x10000 - ((uint32) dd & 0xffff);      /* What's left i
                chunk = c;                                      /* Get the large
                if (chunk > chunks)                             /*   c, chunks, 
                        chunk = chunks;                         /*   0xff00
                if (chunk > chunkd)
                        chunk = chunkd;
                if (chunk > 0xff00)
                        chunk = 0xff00;
                result = memcmp(dd, ss, chunk);
                if (result != 0)
                        return (result);
                dd = (char*) (chunk + (char huge*) dd);
                ss = (char*) (chunk + (char huge*) ss);
                c -= chunk;
        }
        return (0);
}

static void
```

kfax'win3WarningHandler() (./kdegraphics/kfax/libtiffax/tif_win3.c:203)

```
win3WarningHandler(const char* module, const char* fmt, va_list ap)
{
        char e[512] = { '\0' };
        if (module != NULL)
                strcat(strcpy(e, module), ":");
        vsprintf(e+strlen(e), fmt, ap);
        strcat(e, ".");
        MessageBox(GetActiveWindow(), e, "LibTIFF Warning",
            MB_OK|MB_ICONEXCLAMATION);
}
```

kfax'win3ErrorHandler() (./kdegraphics/kfax/libtiffax/tif_win3.c:216)

```
win3ErrorHandler(const char* module, const char* fmt, va_list ap)
{
        char e[512] = { '\0' };
```

```
        if (module != NULL)
                strcat(strcpy(e, module), ":");
        vsprintf(e+strlen(e), fmt, ap);
        strcat(e, ".");
        MessageBox(GetActiveWindow(), e, "LibTIFF Error", MB_OK|MB_ICONSTOP);
}
```

---

kfax'_tiffReadProc() (./kdegraphics/kfax/libtiffax/tif_win32.c:35)

```
_tiffReadProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        DWORD dwSizeRead;
        if (!ReadFile(fd, buf, size, &dwSizeRead, NULL))
                return(0);
        return ((tsize_t) dwSizeRead);
}

static tsize_t
```

---

kfax'_tiffWriteProc() (./kdegraphics/kfax/libtiffax/tif_win32.c:44)

```
_tiffWriteProc(thandle_t fd, tdata_t buf, tsize_t size)
{
        DWORD dwSizeWritten;
        if (!WriteFile(fd, buf, size, &dwSizeWritten, NULL))
                return(0);
        return ((tsize_t) dwSizeWritten);
}

static toff_t
```

---

kfax'_tiffSeekProc() (./kdegraphics/kfax/libtiffax/tif_win32.c:53)

```
_tiffSeekProc(thandle_t fd, toff_t off, int whence)
{
        DWORD dwMoveMethod;
        switch(whence)
        {
        case 0:
                dwMoveMethod = FILE_BEGIN;
                break;
        case 1:
                dwMoveMethod = FILE_CURRENT;
                break;
        case 2:
                dwMoveMethod = FILE_END;
                break;
        default:
                dwMoveMethod = FILE_BEGIN;
                break;
        }
        return ((toff_t)SetFilePointer(fd, off, NULL, dwMoveMethod));
}
```

```
static int
```

---

kfax'_tiffCloseProc() (./kdegraphics/kfax/libtiffax/tif_win32.c:75)

```
_tiffCloseProc(thandle_t fd)
{
        return (CloseHandle(fd) ? 0 : -1);
}

static toff_t
```

---

kfax'_tiffSizeProc() (./kdegraphics/kfax/libtiffax/tif_win32.c:81)

```
_tiffSizeProc(thandle_t fd)
{
        return ((toff_t)GetFileSize(fd, NULL));
}

/*
 * Because Windows uses both a handle and a pointer for file mapping, and only
 * the pointer is returned, the handle must be saved for later use (by the
 * unmap function).  To do this, the tiff structure has an extra member,
 * pv_map_handle, which is contiguous with (4 bytes or one 32-bit word above)
 * the tif_base parameter which is passed as *pbase to the map function.
 * pv_map_handle is then accessed indirectly (and perhaps somewhat unsafely)
 * as an offset from the *pbase parameter by _tiffMapProc.  The handle thus
 * created and saved is destroyed by _tiffUnmapProc, which does not need size
 * in Win32 but receives the map handle value in the size parameter instead.
 */

#pragma argsused
static int
```

---

kfax'_tiffDummyMapProc() (./kdegraphics/kfax/libtiffax/tif_win32.c:100)

```
_tiffDummyMapProc(thandle_t fd, tdata_t* pbase, toff_t* psize)
{
        return(0);
}

static int
```

---

kfax'_tiffMapProc() (./kdegraphics/kfax/libtiffax/tif_win32.c:106)

```
_tiffMapProc(thandle_t fd, tdata_t* pbase, toff_t* psize)
{
        toff_t size;
        HANDLE *phMapFile;
        if ((size = _tiffSizeProc(fd)) == (toff_t)-1)
                return(0);
```

```
        phMapFile = (HANDLE *)(((BYTE *)pbase) + 4);
        if ((*phMapFile = CreateFileMapping(fd, NULL, PAGE_READONLY, 0, size, NU
                        == NULL)
                return(0);
        if ((*pbase = MapViewOfFile(*phMapFile, FILE_MAP_READ, 0, 0, 0)) ==
                        NULL)
        {
                CloseHandle(*phMapFile);
                *phMapFile = NULL;
                return(0);
        }
        *psize = size;
        return(1);
}

#pragma argsused
static void
```

---

kfax'_tiffDummyUnmapProc() (./kdegraphics/kfax/libtiffax/tif_win32.c:129)

```
_tiffDummyUnmapProc(thandle_t fd, tdata_t base, toff_t size)
{
        return;
}

static void
```

---

kfax'_tiffUnmapProc() (./kdegraphics/kfax/libtiffax/tif_win32.c:135)

```
_tiffUnmapProc(thandle_t fd, tdata_t base, toff_t map_handle)
{
        UnmapViewOfFile(base);
        CloseHandle((HANDLE)map_handle);
        return;
}

/*
 * Open a TIFF file descriptor for read/writing.
 * Note that TIFFFdOpen and TIFFOpen recognise the character 'u' in the mode
 * string, which forces the file to be opened unmapped.
 */
TIFF*
```

---

kfax'TIFFFdOpen() (./kdegraphics/kfax/libtiffax/tif_win32.c:148)

```
TIFFFdOpen(int ifd, const char* name, const char* mode)
{
        TIFF* tif;
        BOOL fSuppressMap = (mode[1] == 'u' || mode[2] == 'u');

        tif = TIFFClientOpen(name, mode,
                  (thandle_t)ifd,
            _tiffReadProc, _tiffWriteProc,
```

```
            _tiffSeekProc, _tiffCloseProc, _tiffSizeProc,
                 fSuppressMap ? _tiffDummyMapProc : _tiffMapProc,
                 fSuppressMap ? _tiffDummyUnmapProc : _tiffUnmapProc);
        if (tif)
                tif->tif_fd = ifd;
        return (tif);
}

/*
 * Open a TIFF file for read/writing.
 */
TIFF*
```

---

kfax'TIFFOpen() (./kdegraphics/kfax/libtiffax/tif_win32.c:168)

```
TIFFOpen(const char* name, const char* mode)
{
        static const char module[] = "TIFFOpen";
        thandle_t fd;
        int m;
        DWORD dwMode;

        m = _TIFFgetMode(mode, module);

        switch(m)
        {
        case O_RDONLY:
                dwMode = OPEN_EXISTING;
                break;
        case O_RDWR:
                dwMode = OPEN_ALWAYS;
                break;
        case O_RDWR|O_CREAT:
                dwMode = CREATE_NEW;
                break;
        case O_RDWR|O_TRUNC:
                dwMode = CREATE_ALWAYS;
                break;
        case O_RDWR|O_CREAT|O_TRUNC:
                dwMode = CREATE_ALWAYS;
                break;
        default:
                return ((TIFF*)0);
        }
        fd = (thandle_t)CreateFile(name, (m == O_RDONLY) ? GENERIC_READ :
                        (GENERIC_READ | GENERIC_WRITE), FILE_SHARE_READ, NULL, dw
                        (m == O_RDONLY) ? FILE_ATTRIBUTE_READONLY : FILE_ATTRIBU
        if (fd == INVALID_HANDLE_VALUE) {
                TIFFError(module, "%s: Cannot open", name);
                return ((TIFF *)0);
        }
        return (TIFFFdOpen((int)fd, name, mode));
}

tdata_t
```

---

kfax'_TIFFmalloc() (./kdegraphics/kfax/libtiffax/tif_win32.c:208)

```
_TIFFmalloc(tsize_t s)
{
        return ((tdata_t)GlobalAlloc(GMEM_FIXED, s));
}

void
```

---

kfax'_TIFFfree() (./kdegraphics/kfax/libtiffax/tif_win32.c:214)

```
_TIFFfree(tdata_t p)
{
        GlobalFree(p);
        return;
}

tdata_t
```

---

kfax'_TIFFrealloc() (./kdegraphics/kfax/libtiffax/tif_win32.c:221)

```
_TIFFrealloc(tdata_t p, tsize_t s)
{
        void* pvTmp;
        if ((pvTmp = GlobalReAlloc(p, s, 0)) == NULL) {
                if ((pvTmp = GlobalAlloc(GMEM_FIXED, s)) != NULL) {
                        CopyMemory(pvTmp, p, s);
                        GlobalFree(p);
                }
        }
        return ((tdata_t)pvTmp);
}

void
```

---

kfax'_TIFFmemset() (./kdegraphics/kfax/libtiffax/tif_win32.c:234)

```
_TIFFmemset(void* p, int v, tsize_t c)
{
        FillMemory(p, c, (BYTE)v);
}

void
```

---

kfax'_TIFFmemcpy() (./kdegraphics/kfax/libtiffax/tif_win32.c:240)

```
_TIFFmemcpy(void* d, const tdata_t s, tsize_t c)
{
        CopyMemory(d, s, c);
```

```
}

int
```

---

kfax'_TIFFmemcmp() (./kdegraphics/kfax/libtiffax/tif_win32.c:246)

```
_TIFFmemcmp(const tdata_t p1, const tdata_t p2, tsize_t c)
{
        register const BYTE *pb1 = p1;
        register const BYTE *pb2 = p2;
        register DWORD dwTmp = c;
        register int iTmp;
        for (iTmp = 0; dwTmp-- && !iTmp; iTmp = (int)*pb1++ - (int)*pb2++)
                ;
        return (iTmp);
}

static void
```

---

kfax'Win32WarningHandler() (./kdegraphics/kfax/libtiffax/tif_win32.c:258)

```
Win32WarningHandler(const char* module, const char* fmt, va_list ap)
{
        LPTSTR szTitle;
        LPTSTR szTmp;
        LPCTSTR szTitleText = "%s Warning";
        LPCTSTR szDefaultModule = "TIFFLIB";
        szTmp = (module == NULL) ? (LPTSTR)szDefaultModule : (LPTSTR)module;
        if ((szTitle = (LPTSTR)LocalAlloc(LMEM_FIXED, (lstrlen(szTmp) +
                        lstrlen(szTitleText) + lstrlen(fmt) + 128)*sizeof(TCHAR)
                return;
        wsprintf(szTitle, szTitleText, szTmp);
        szTmp = szTitle + (lstrlen(szTitle)+2)*sizeof(TCHAR);
        wvsprintf(szTmp, fmt, ap);
        MessageBox(GetFocus(), szTmp, szTitle, MB_OK | MB_ICONINFORMATION);
        LocalFree(szTitle);
        return;
}
```

---

kfax'Win32ErrorHandler() (./kdegraphics/kfax/libtiffax/tif_win32.c:278)

```
Win32ErrorHandler(const char* module, const char* fmt, va_list ap)
{
        LPTSTR szTitle;
        LPTSTR szTmp;
        LPCTSTR szTitleText = "%s Error";
        LPCTSTR szDefaultModule = "TIFFLIB";
        szTmp = (module == NULL) ? (LPTSTR)szDefaultModule : (LPTSTR)module;
        if ((szTitle = (LPTSTR)LocalAlloc(LMEM_FIXED, (lstrlen(szTmp) +
                        lstrlen(szTitleText) + lstrlen(fmt) + 128)*sizeof(TCHAR)
                return;
        wsprintf(szTitle, szTitleText, szTmp);
        szTmp = szTitle + (lstrlen(szTitle)+2)*sizeof(TCHAR);
```

```
        wvsprintf(szTmp, fmt, ap);
        MessageBox(GetFocus(), szTmp, szTitle, MB_OK | MB_ICONEXCLAMATION);
        LocalFree(szTitle);
        return;
}
```

---

kfax'TIFFWriteScanline() (./kdegraphics/kfax/libtiffax/tif_write.c:52)

```
TIFFWriteScanline(TIFF* tif, tdata_t buf, uint32 row, tsample_t sample)
{
        static const char module[] = "TIFFWriteScanline";
        register TIFFDirectory *td;
        int status, imagegrew = 0;
        tstrip_t strip;

        if (!WRITECHECKSTRIPS(tif, module))
                return (-1);
        /*
         * Handle delayed allocation of data buffer.  This
         * permits it to be sized more intelligently (using
         * directory information).
         */
        if (!BUFFERCHECK(tif))
                return (-1);
        td = &tif->tif_dir;
        /*
         * Extend image length if needed
         * (but only for PlanarConfig=1).
         */
        if (row >= td->td_imagelength) {        /* extend image */
                if (td->td_planarconfig == PLANARCONFIG_SEPARATE) {
                        TIFFError(tif->tif_name,
                "Can not change \"ImageLength\" when using separate planes");
                        return (-1);
                }
                td->td_imagelength = row+1;
                imagegrew = 1;
        }
        /*
         * Calculate strip and check for crossings.
         */
        if (td->td_planarconfig == PLANARCONFIG_SEPARATE) {
                if (sample >= td->td_samplesperpixel) {
                        TIFFError(tif->tif_name,
                            "%d: Sample out of range, max %d",
                            sample, td->td_samplesperpixel);
                        return (-1);
                }
                strip = sample*td->td_stripsperimage + row/td->td_rowsperstrip;
        } else
                strip = row / td->td_rowsperstrip;
        if (strip != tif->tif_curstrip) {
                /*
                 * Changing strips -- flush any data present.
                 */
                if (!TIFFFlushData(tif))
                        return (-1);
                tif->tif_curstrip = strip;
```

```
                /*
                 * Watch out for a growing image.  The value of
                 * strips/image will initially be 1 (since it
                 * can't be deduced until the imagelength is known).
                 */
                if (strip >= td->td_stripsperimage && imagegrew)
                        td->td_stripsperimage =
                            TIFFhowmany(td->td_imagelength,td->td_rowsperstrip);
                tif->tif_row =
                    (strip % td->td_stripsperimage) * td->td_rowsperstrip;
                if ((tif->tif_flags & TIFF_CODERSETUP) == 0) {
                        if (!(*tif->tif_setupencode)(tif))
                                return (-1);
                        tif->tif_flags |= TIFF_CODERSETUP;
                }
                if (!(*tif->tif_preencode)(tif, sample))
                        return (-1);
                tif->tif_flags |= TIFF_POSTENCODE;
        }
        /*
         * Check strip array to make sure there's space.
         * We don't support dynamically growing files that
         * have data organized in separate bitplanes because
         * it's too painful.  In that case we require that
         * the imagelength be set properly before the first
         * write (so that the strips array will be fully
         * allocated above).
         */
        if (strip >= td->td_nstrips && !TIFFGrowStrips(tif, 1, module))
                return (-1);
        /*
         * Ensure the write is either sequential or at the
         * beginning of a strip (or that we can randomly
         * access the data -- i.e. no encoding).
         */
        if (row != tif->tif_row) {
                if (row < tif->tif_row) {
                        /*
                         * Moving backwards within the same strip:
                         * backup to the start and then decode
                         * forward (below).
                         */
                        tif->tif_row = (strip % td->td_stripsperimage) *
                            td->td_rowsperstrip;
                        tif->tif_rawcp = tif->tif_rawdata;
                }
                /*
                 * Seek forward to the desired row.
                 */
                if (!(*tif->tif_seek)(tif, row - tif->tif_row))
                        return (-1);
                tif->tif_row = row;
        }
        status = (*tif->tif_encoderow)(tif, (tidata_t) buf,
            tif->tif_scanlinesize, sample);
        tif->tif_row++;
        return (status);
}

/*
 * Encode the supplied data and write it to the
```

```
 * specified strip.  There must be space for the
 * data; we don't check if strips overlap!
 *
 * NB: Image length must be setup before writing.
 */
tsize_t
```

---

kfax'TIFFWriteEncodedStrip() (./kdegraphics/kfax/libtiffax/tif_write.c:169)

```
TIFFWriteEncodedStrip(TIFF* tif, tstrip_t strip, tdata_t data, tsize_t cc)
{
        static const char module[] = "TIFFWriteEncodedStrip";
        TIFFDirectory *td = &tif->tif_dir;
        tsample_t sample;

        if (!WRITECHECKSTRIPS(tif, module))
                return ((tsize_t) -1);
        /*
         * Check strip array to make sure there's space.
         * We don't support dynamically growing files that
         * have data organized in separate bitplanes because
         * it's too painful.  In that case we require that
         * the imagelength be set properly before the first
         * write (so that the strips array will be fully
         * allocated above).
         */
        if (strip >= td->td_nstrips) {
                if (td->td_planarconfig == PLANARCONFIG_SEPARATE) {
                        TIFFError(tif->tif_name,
                "Can not grow image by strips when using separate planes");
                        return ((tsize_t) -1);
                }
                if (!TIFFGrowStrips(tif, 1, module))
                        return ((tsize_t) -1);
                td->td_stripsperimage =
                    TIFFhowmany(td->td_imagelength, td->td_rowsperstrip);
        }
        /*
         * Handle delayed allocation of data buffer.  This
         * permits it to be sized according to the directory
         * info.
         */
        if (!BUFFERCHECK(tif))
                return ((tsize_t) -1);
        tif->tif_curstrip = strip;
        tif->tif_row = (strip % td->td_stripsperimage) * td->td_rowsperstrip;
        if ((tif->tif_flags & TIFF_CODERSETUP) == 0) {
                if (!(*tif->tif_setupencode)(tif))
                        return ((tsize_t) -1);
                tif->tif_flags |= TIFF_CODERSETUP;
        }
        tif->tif_flags &= ~TIFF_POSTENCODE;
        sample = (tsample_t)(strip / td->td_stripsperimage);
        if (!(*tif->tif_preencode)(tif, sample))
                return ((tsize_t) -1);
        if (!(*tif->tif_encodestrip)(tif, (tidata_t) data, cc, sample))
                return ((tsize_t) 0);
        if (!(*tif->tif_postencode)(tif))
```

```
                return (-1);
        if (!isFillOrder(tif, td->td_fillorder) &&
            (tif->tif_flags & TIFF_NOBITREV) == 0)
                TIFFReverseBits(tif->tif_rawdata, tif->tif_rawcc);
        if (tif->tif_rawcc > 0 &&
            !TIFFAppendToStrip(tif, strip, tif->tif_rawdata, tif->tif_rawcc))
                return (-1);
        tif->tif_rawcc = 0;
        tif->tif_rawcp = tif->tif_rawdata;
        return (cc);
}

/*
 * Write the supplied data to the specified strip.
 * There must be space for the data; we don't check
 * if strips overlap!
 *
 * NB: Image length must be setup before writing.
 */
tsize_t
```

---

kfax'TIFFWriteRawStrip() (./kdegraphics/kfax/libtiffax/tif_write.c:238)

```
TIFFWriteRawStrip(TIFF* tif, tstrip_t strip, tdata_t data, tsize_t cc)
{
        static const char module[] = "TIFFWriteRawStrip";
        TIFFDirectory *td = &tif->tif_dir;

        if (!WRITECHECKSTRIPS(tif, module))
                return ((tsize_t) -1);
        /*
         * Check strip array to make sure there's space.
         * We don't support dynamically growing files that
         * have data organized in separate bitplanes because
         * it's too painful.  In that case we require that
         * the imagelength be set properly before the first
         * write (so that the strips array will be fully
         * allocated above).
         */
        if (strip >= td->td_nstrips) {
                if (td->td_planarconfig == PLANARCONFIG_SEPARATE) {
                        TIFFError(tif->tif_name,
                "Can not grow image by strips when using separate planes");
                        return ((tsize_t) -1);
                }
                /*
                 * Watch out for a growing image.  The value of
                 * strips/image will initially be 1 (since it
                 * can't be deduced until the imagelength is known).
                 */
                if (strip >= td->td_stripsperimage)
                        td->td_stripsperimage =
                            TIFFhowmany(td->td_imagelength,td->td_rowsperstrip);
                if (!TIFFGrowStrips(tif, 1, module))
                        return ((tsize_t) -1);
        }
        tif->tif_curstrip = strip;
        tif->tif_row = (strip % td->td_stripsperimage) * td->td_rowsperstrip;
```

```
        return (TIFFAppendToStrip(tif, strip, (tidata_t) data, cc) ?
            cc : (tsize_t) -1);
}

/*
 * Write and compress a tile of data.  The
 * tile is selected by the (x,y,z,s) coordinates.
 */
tsize_t
```

---

kfax'TIFFWriteTile() (./kdegraphics/kfax/libtiffax/tif_write.c:282)

```
TIFFWriteTile(TIFF* tif,
    tdata_t buf, uint32 x, uint32 y, uint32 z, tsample_t s)
{
        if (!TIFFCheckTile(tif, x, y, z, s))
                return (-1);
        /*
         * NB: A tile size of -1 is used instead of tif_tilesize knowing
         *     that TIFFWriteEncodedTile will clamp this to the tile size.
         *     This is done because the tile size may not be defined until
         *     after the output buffer is setup in TIFFWriteBufferSetup.
         */
        return (TIFFWriteEncodedTile(tif,
            TIFFComputeTile(tif, x, y, z, s), buf, (tsize_t) -1));
}

/*
 * Encode the supplied data and write it to the
 * specified tile.  There must be space for the
 * data.  The function clamps individual writes
 * to a tile to the tile size, but does not (and
 * can not) check that multiple writes to the same
 * tile do not write more than tile size data.
 *
 * NB: Image length must be setup before writing; this
 *     interface does not support automatically growing
 *     the image on each write (as TIFFWriteScanline does).
 */
tsize_t
```

---

kfax'TIFFWriteEncodedTile() (./kdegraphics/kfax/libtiffax/tif_write.c:310)

```
TIFFWriteEncodedTile(TIFF* tif, ttile_t tile, tdata_t data, tsize_t cc)
{
        static const char module[] = "TIFFWriteEncodedTile";
        TIFFDirectory *td;
        tsample_t sample;

        if (!WRITECHECKTILES(tif, module))
                return ((tsize_t) -1);
        td = &tif->tif_dir;
        if (tile >= td->td_nstrips) {
                TIFFError(module, "%s: Tile %lu out of range, max %lu",
                    tif->tif_name, (u_long) tile, (u_long) td->td_nstrips);
                return ((tsize_t) -1);
```

```
	}
	/*
	 * Handle delayed allocation of data buffer.  This
	 * permits it to be sized more intelligently (using
	 * directory information).
	 */
	if (!BUFFERCHECK(tif))
		return ((tsize_t) -1);
	tif->tif_curtile = tile;
	/*
	 * Compute tiles per row & per column to compute
	 * current row and column
	 */
	tif->tif_row = (tile % TIFFhowmany(td->td_imagelength, td->td_tilelength
		* td->td_tilelength;
	tif->tif_col = (tile % TIFFhowmany(td->td_imagewidth, td->td_tilewidth))
		* td->td_tilewidth;

	if ((tif->tif_flags & TIFF_CODERSETUP) == 0) {
		if (!(*tif->tif_setupencode)(tif))
			return ((tsize_t) -1);
		tif->tif_flags |= TIFF_CODERSETUP;
	}
	tif->tif_flags &= ~TIFF_POSTENCODE;
	sample = (tsample_t)(tile/td->td_stripsperimage);
	if (!(*tif->tif_preencode)(tif, sample))
		return ((tsize_t) -1);
	/*
	 * Clamp write amount to the tile size.  This is mostly
	 * done so that callers can pass in some large number
	 * (e.g. -1) and have the tile size used instead.
	 */
	if ((uint32) cc > tif->tif_tilesize)
		cc = tif->tif_tilesize;
	if (!(*tif->tif_encodetile)(tif, (tidata_t) data, cc, sample))
		return ((tsize_t) 0);
	if (!(*tif->tif_postencode)(tif))
		return ((tsize_t) -1);
	if (!isFillOrder(tif, td->td_fillorder) &&
	    (tif->tif_flags & TIFF_NOBITREV) == 0)
		TIFFReverseBits((u_char *)tif->tif_rawdata, tif->tif_rawcc);
	if (tif->tif_rawcc > 0 && !TIFFAppendToStrip(tif, tile,
	    tif->tif_rawdata, tif->tif_rawcc))
		return ((tsize_t) -1);
	tif->tif_rawcc = 0;
	tif->tif_rawcp = tif->tif_rawdata;
	return (cc);
}

/*
 * Write the supplied data to the specified strip.
 * There must be space for the data; we don't check
 * if strips overlap!
 *
 * NB: Image length must be setup before writing; this
 *     interface does not support automatically growing
 *     the image on each write (as TIFFWriteScanline does).
 */
tsize_t
```

kfax'TIFFWriteRawTile() (./kdegraphics/kfax/libtiffax/tif_write.c:382)

```
TIFFWriteRawTile(TIFF* tif, ttile_t tile, tdata_t data, tsize_t cc)
{
        static const char module[] = "TIFFWriteRawTile";

        if (!WRITECHECKTILES(tif, module))
                return ((tsize_t) -1);
        if (tile >= tif->tif_dir.td_nstrips) {
                TIFFError(module, "%s: Tile %lu out of range, max %lu",
                    tif->tif_name, (u_long) tile,
                    (u_long) tif->tif_dir.td_nstrips);
                return ((tsize_t) -1);
        }
        return (TIFFAppendToStrip(tif, tile, (tidata_t) data, cc) ?
            cc : (tsize_t) -1);
}
```

---

kfax'TIFFSetupStrips() (./kdegraphics/kfax/libtiffax/tif_write.c:402)

```
TIFFSetupStrips(TIFF* tif)
{
        TIFFDirectory* td = &tif->tif_dir;

        if (isTiled(tif))
                td->td_stripsperimage = isUnspecified(td, td_tilelength) ?
                    td->td_samplesperpixel : TIFFNumberOfTiles(tif);
        else
                td->td_stripsperimage = isUnspecified(td, td_rowsperstrip) ?
                    td->td_samplesperpixel : TIFFNumberOfStrips(tif);
        td->td_nstrips = td->td_stripsperimage;
        if (td->td_planarconfig == PLANARCONFIG_SEPARATE)
                td->td_stripsperimage /= td->td_samplesperpixel;
        td->td_stripoffset = (uint32 *)
            _TIFFmalloc(td->td_nstrips * sizeof (uint32));
        td->td_stripbytecount = (uint32 *)
            _TIFFmalloc(td->td_nstrips * sizeof (uint32));
        if (td->td_stripoffset == NULL || td->td_stripbytecount == NULL)
                return (0);
        /*
         * Place data at the end-of-file
         * (by setting offsets to zero).
         */
        _TIFFmemset(td->td_stripoffset, 0, td->td_nstrips*sizeof (uint32));
        _TIFFmemset(td->td_stripbytecount, 0, td->td_nstrips*sizeof (uint32));
        TIFFSetFieldBit(tif, FIELD_STRIPOFFSETS);
        TIFFSetFieldBit(tif, FIELD_STRIPBYTECOUNTS);
        return (1);
}
```

---

kfax'TIFFWriteCheck() (./kdegraphics/kfax/libtiffax/tif_write.c:440)

```
TIFFWriteCheck(TIFF* tif, int tiles, const char* module)
```

```
{
        if (tif->tif_mode == O_RDONLY) {
                TIFFError(module, "%s: File not open for writing",
                    tif->tif_name);
                return (0);
        }
        if (tiles ^ isTiled(tif)) {
                TIFFError(tif->tif_name, tiles ?
                    "Can not write tiles to a stripped image" :
                    "Can not write scanlines to a tiled image");
                return (0);
        }
        /*
         * On the first write verify all the required information
         * has been setup and initialize any data structures that
         * had to wait until directory information was set.
         * Note that a lot of our work is assumed to remain valid
         * because we disallow any of the important parameters
         * from changing after we start writing (i.e. once
         * TIFF_BEENWRITING is set, TIFFSetField will only allow
         * the image's length to be changed).
         */
        if (!TIFFFieldSet(tif, FIELD_IMAGEDIMENSIONS)) {
                TIFFError(module,
                    "%s: Must set \"ImageWidth\" before writing data",
                    tif->tif_name);
                return (0);
        }
        if (!TIFFFieldSet(tif, FIELD_PLANARCONFIG)) {
                TIFFError(module,
            "%s: Must set \"PlanarConfiguration\" before writing data",
                    tif->tif_name);
                return (0);
        }
        if (tif->tif_dir.td_stripoffset == NULL && !TIFFSetupStrips(tif)) {
                tif->tif_dir.td_nstrips = 0;
                TIFFError(module, "%s: No space for %s arrays",
                    tif->tif_name, isTiled(tif) ? "tile" : "strip");
                return (0);
        }
        tif->tif_tilesize = TIFFTileSize(tif);
        tif->tif_scanlinesize = TIFFScanlineSize(tif);
        tif->tif_flags |= TIFF_BEENWRITING;
        return (1);
}

/*
 * Setup the raw data buffer used for encoding.
 */
int
```

---

kfax'TIFFWriteBufferSetup() (./kdegraphics/kfax/libtiffax/tif_write.c:491)

```
TIFFWriteBufferSetup(TIFF* tif, tdata_t bp, tsize_t size)
{
        static const char module[] = "TIFFWriteBufferSetup";

        if (tif->tif_rawdata) {
```

```
                if (tif->tif_flags & TIFF_MYBUFFER) {
                        _TIFFfree(tif->tif_rawdata);
                        tif->tif_flags &= ~TIFF_MYBUFFER;
                }
                tif->tif_rawdata = NULL;
        }
        if (size == (tsize_t) -1) {
                size = (isTiled(tif) ?
                    tif->tif_tilesize : tif->tif_scanlinesize);
                /*
                 * Make raw data buffer at least 8K
                 */
                if (size < 8*1024)
                        size = 8*1024;
                bp = NULL;                      /* NB: force malloc */
        }
        if (bp == NULL) {
                bp = _TIFFmalloc(size);
                if (bp == NULL) {
                        TIFFError(module, "%s: No space for output buffer",
                            tif->tif_name);
                        return (0);
                }
                tif->tif_flags |= TIFF_MYBUFFER;
        } else
                tif->tif_flags &= ~TIFF_MYBUFFER;
        tif->tif_rawdata = (tidata_t) bp;
        tif->tif_rawdatasize = size;
        tif->tif_rawcc = 0;
        tif->tif_rawcp = tif->tif_rawdata;
        tif->tif_flags |= TIFF_BUFFERSETUP;
        return (1);
}

/*
 * Grow the strip data structures by delta strips.
 */
static int
```

---

kfax'TIFFGrowStrips() (./kdegraphics/kfax/libtiffax/tif_write.c:534)

```
TIFFGrowStrips(TIFF* tif, int delta, const char* module)
{
        TIFFDirectory *td = &tif->tif_dir;

        assert(td->td_planarconfig == PLANARCONFIG_CONTIG);
        td->td_stripoffset = (uint32*)_TIFFrealloc(td->td_stripoffset,
            (td->td_nstrips + delta) * sizeof (uint32));
        td->td_stripbytecount = (uint32*)_TIFFrealloc(td->td_stripbytecount,
            (td->td_nstrips + delta) * sizeof (uint32));
        if (td->td_stripoffset == NULL || td->td_stripbytecount == NULL) {
                td->td_nstrips = 0;
                TIFFError(module, "%s: No space to expand strip arrays",
                    tif->tif_name);
                return (0);
        }
        _TIFFmemset(td->td_stripoffset+td->td_nstrips, 0, delta*sizeof (uint32))
        _TIFFmemset(td->td_stripbytecount+td->td_nstrips, 0, delta*sizeof (uint3
```

```
        td->td_nstrips += delta;
        return (1);
}

/*
 * Append the data to the specified strip.
 *
 * NB: We don't check that there's space in the
 *     file (i.e. that strips do not overlap).
 */
static int
```

---

kfax'TIFFAppendToStrip() (./kdegraphics/kfax/libtiffax/tif_write.c:562)

```
TIFFAppendToStrip(TIFF* tif, tstrip_t strip, tidata_t data, tsize_t cc)
{
        TIFFDirectory *td = &tif->tif_dir;
        static const char module[] = "TIFFAppendToStrip";

        if (td->td_stripoffset[strip] == 0 || tif->tif_curoff == 0) {
                /*
                 * No current offset, set the current strip.
                 */
                if (td->td_stripoffset[strip] != 0) {
                        if (!SeekOK(tif, td->td_stripoffset[strip])) {
                                TIFFError(module,
                                    "%s: Seek error at scanline %lu",
                                    tif->tif_name, (u_long) tif->tif_row);
                                return (0);
                        }
                } else
                        td->td_stripoffset[strip] =
                            TIFFSeekFile(tif, (toff_t) 0, SEEK_END);
                tif->tif_curoff = td->td_stripoffset[strip];
        }
        if (!WriteOK(tif, data, cc)) {
                TIFFError(module, "%s: Write error at scanline %lu",
                    tif->tif_name, (u_long) tif->tif_row);
                return (0);
        }
        tif->tif_curoff += cc;
        td->td_stripbytecount[strip] += cc;
        return (1);
}

/*
 * Internal version of TIFFFlushData that can be
 * called by ``encodestrip routines'' w/o concern
 * for infinite recursion.
 */
int
```

---

kfax'TIFFFlushData1() (./kdegraphics/kfax/libtiffax/tif_write.c:599)

```
TIFFFlushData1(TIFF* tif)
{
```

```
        if (tif->tif_rawcc > 0) {
                if (!isFillOrder(tif, tif->tif_dir.td_fillorder) &&
                    (tif->tif_flags & TIFF_NOBITREV) == 0)
                        TIFFReverseBits((u_char *)tif->tif_rawdata,
                            tif->tif_rawcc);
                if (!TIFFAppendToStrip(tif,
                    isTiled(tif) ? tif->tif_curtile : tif->tif_curstrip,
                    tif->tif_rawdata, tif->tif_rawcc))
                        return (0);
                tif->tif_rawcc = 0;
                tif->tif_rawcp = tif->tif_rawdata;
        }
        return (1);
}

/*
 * Set the current write offset.  This should only be
 * used to set the offset to a known previous location
 * (very carefully), or to 0 so that the next write gets
 * appended to the end of the file.
 */
void
```

---

kfax'TIFFSetWriteOffset() (./kdegraphics/kfax/libtiffax/tif_write.c:623)

```
TIFFSetWriteOffset(TIFF* tif, toff_t off)
{
        tif->tif_curoff = off;
}
```

---

kfax'ZIPSetupDecode() (./kdegraphics/kfax/libtiffax/tif_zip.c:64)

```
ZIPSetupDecode(TIFF* tif)
{
        ZIPState* sp = DecoderState(tif);
        static char module[] = "ZIPSetupDecode";

        assert(sp != NULL);
        if (inflateInit2(&sp->stream, -DEF_WBITS) != Z_OK) {
                TIFFError(module, "%s: %s", tif->tif_name, sp->stream.msg);
                return (0);
        } else
                return (1);
}

/*
 * Setup state for decoding a strip.
 */
static int
```

---

kfax'ZIPPreDecode() (./kdegraphics/kfax/libtiffax/tif_zip.c:81)

```
ZIPPreDecode(TIFF* tif, tsample_t s)
```

```
{
        ZIPState* sp = DecoderState(tif);

        (void) s;
        assert(sp != NULL);
        sp->stream.next_in = tif->tif_rawdata;
        sp->stream.avail_in = tif->tif_rawcc;
        return (inflateReset(&sp->stream) == Z_OK);
}

static int
```

---

kfax'ZIPDecode() (./kdegraphics/kfax/libtiffax/tif_zip.c:93)

```
ZIPDecode(TIFF* tif, tidata_t op, tsize_t occ, tsample_t s)
{
        ZIPState* sp = DecoderState(tif);
        static char module[] = "ZIPDecode";

        (void) s;
        assert(sp != NULL);
        sp->stream.next_out = op;
        sp->stream.avail_out = occ;
        do {
                int state = inflate(&sp->stream, Z_PARTIAL_FLUSH);
                if (state == Z_STREAM_END)
                        break;
                if (state == Z_DATA_ERROR) {
                        TIFFError(module,
                            "%s: Decoding error at scanline %d, %s",
                            tif->tif_name, tif->tif_row, sp->stream.msg);
                        if (inflateSync(&sp->stream) != Z_OK)
                                return (0);
                        continue;
                }
                if (state != Z_OK) {
                        TIFFError(module, "%s: libgz error: %s",
                            tif->tif_name, sp->stream.msg);
                        return (0);
                }
        } while (sp->stream.avail_out > 0);
        if (sp->stream.avail_out != 0) {
                TIFFError(module,
                    "%s: Not enough data at scanline %d (short %d bytes)",
                    tif->tif_name, tif->tif_row, sp->stream.avail_out);
                return (0);
        }
        return (1);
}

static int
```

---

kfax'ZIPSetupEncode() (./kdegraphics/kfax/libtiffax/tif_zip.c:130)

```
ZIPSetupEncode(TIFF* tif)
{
```

```
        ZIPState* sp = EncoderState(tif);
        static char module[] = "ZIPSetupEncode";

        assert(sp != NULL);
        /*
         * We use the undocumented feature of a negiatve window
         * bits to suppress writing the header in the output
         * stream.  This is necessary when the resulting image
         * is made up of multiple strips or tiles as otherwise
         * libgz will not write a header for each strip/tile and
         * the decoder will fail.
         */
        if (deflateInit2(&sp->stream, Z_DEFAULT_COMPRESSION,
            DEFLATED, -MAX_WBITS, DEF_MEM_LEVEL, 0) != Z_OK) {
                TIFFError(module, "%s: %s", tif->tif_name, sp->stream.msg);
                return (0);
        } else
                return (1);
}

/*
 * Reset encoding state at the start of a strip.
 */
static int
```

---

kfax'ZIPPreEncode() (./kdegraphics/kfax/libtiffax/tif_zip.c:156)

```
ZIPPreEncode(TIFF* tif, tsample_t s)
{
        ZIPState *sp = EncoderState(tif);

        (void) s;
        assert(sp != NULL);
        sp->stream.next_out = tif->tif_rawdata;
        sp->stream.avail_out = tif->tif_rawdatasize;
        return (deflateReset(&sp->stream) == Z_OK);
}

/*
 * Encode a chunk of pixels.
 */
static int
```

---

kfax'ZIPEncode() (./kdegraphics/kfax/libtiffax/tif_zip.c:171)

```
ZIPEncode(TIFF* tif, tidata_t bp, tsize_t cc, tsample_t s)
{
        ZIPState *sp = EncoderState(tif);
        static char module[] = "ZIPEncode";

        (void) s;
        sp->stream.next_in = bp;
        sp->stream.avail_in = cc;
        do {
                if (deflate(&sp->stream, Z_NO_FLUSH) != Z_OK) {
                        TIFFError(module, "%s: Encoder error: %s",
```

```
                                tif->tif_name, sp->stream.msg);
                        return (0);
                }
                if (sp->stream.avail_out == 0) {
                        tif->tif_rawcc = tif->tif_rawdatasize;
                        TIFFFlushData1(tif);
                        sp->stream.next_out = tif->tif_rawdata;
                        sp->stream.avail_out = tif->tif_rawdatasize;
                }
        } while (sp->stream.avail_in > 0);
        return (1);
}

/*
 * Finish off an encoded strip by flushing the last
 * string and tacking on an End Of Information code.
 */
static int
```

---

kfax'ZIPPostEncode() (./kdegraphics/kfax/libtiffax/tif_zip.c:200)

```
ZIPPostEncode(TIFF* tif)
{
        ZIPState *sp = EncoderState(tif);
        static char module[] = "ZIPPostEncode";
        int state;

        sp->stream.avail_in = 0;
        do {
                state = deflate(&sp->stream, Z_FINISH);
                switch (state) {
                case Z_STREAM_END:
                case Z_OK:
                    if (sp->stream.avail_out != tif->tif_rawdatasize) {
                            tif->tif_rawcc =
                                tif->tif_rawdatasize - sp->stream.avail_out;
                            TIFFFlushData1(tif);
                            sp->stream.next_out = tif->tif_rawdata;
                            sp->stream.avail_out = tif->tif_rawdatasize;
                    }
                    break;
                default:
                    TIFFError(module, "%s: libgz error: %s",
                        tif->tif_name, sp->stream.msg);
                    return (0);
                }
        } while (state != Z_STREAM_END);
        return (1);
}

static void
```

---

kfax'ZIPCleanup() (./kdegraphics/kfax/libtiffax/tif_zip.c:230)

```
ZIPCleanup(TIFF* tif)
{
```

```
        ZIPState* sp = (ZIPState*) tif->tif_data;
        if (sp) {
                if (tif->tif_mode == O_RDONLY)
                        inflateEnd(&sp->stream);
                else
                        deflateEnd(&sp->stream);
                _TIFFfree(sp);
                tif->tif_data = NULL;
        }
}

int
```

---

kfax'TIFFInitZIP() (./kdegraphics/kfax/libtiffax/tif_zip.c:244)

```
TIFFInitZIP(TIFF* tif, int scheme)
{
        ZIPState* sp;

        assert(scheme == COMPRESSION_DEFLATE);

        /*
         * Allocate state block so tag methods have storage to record values.
         */
        tif->tif_data = (tidata_t) _TIFFmalloc(sizeof (ZIPState));
        if (tif->tif_data == NULL)
                goto bad;
        sp = (ZIPState*) tif->tif_data;
        sp->stream.zalloc = NULL;
        sp->stream.zfree = NULL;
        sp->stream.opaque = NULL;
        sp->stream.data_type = Z_BINARY;

        /*
         * Install codec methods.
         */
        tif->tif_setupdecode = ZIPSetupDecode;
        tif->tif_predecode = ZIPPreDecode;
        tif->tif_decoderow = ZIPDecode;
        tif->tif_decodestrip = ZIPDecode;
        tif->tif_decodetile = ZIPDecode;
        tif->tif_setupencode = ZIPSetupEncode;
        tif->tif_preencode = ZIPPreEncode;
        tif->tif_postencode = ZIPPostEncode;
        tif->tif_encoderow = ZIPEncode;
        tif->tif_encodestrip = ZIPEncode;
        tif->tif_encodetile = ZIPEncode;
        tif->tif_cleanup = ZIPCleanup;
        /*
         * Setup predictor setup.
         */
        (void) TIFFPredictorInit(tif);
        return (1);
bad:
        TIFFError("TIFFInitZIP", "No space for ZIP state block");
        return (0);
}
```

---

kfax'tiff2psmain() (./kdegraphics/kfax/tiff2ps.c:84)

```
int tiff2psmain( char* tiff_file,  FILE* psoutput) {



  int dirnum = -1, c, np = 0;
  float pageWidth = 0;
  float pageHeight = 0;
  uint32 diroff = 0;




  TIFF *tif = TIFFOpen(tiff_file, "r");

  if (tif != NULL) {
    if (dirnum != -1 && !TIFFSetDirectory(tif, dirnum))
      return (-1);
    else if (diroff != 0 &&
             !TIFFSetSubDirectory(tif, diroff))
      return (-1);
    np = TIFF2PS(psoutput, tif, pageWidth, pageHeight);
    TIFFClose(tif);
  }

  if (np)
    PSTail(psoutput, np);

  return (0);
}




int
```

---

kfax'tiff2psmainold() (./kdegraphics/kfax/tiff2ps.c:119)

```
tiff2psmainold(int argc, char* argv[])
{



        int dirnum = -1, c, np = 0;
        float pageWidth = 0;
        float pageHeight = 0;
        uint32 diroff = 0;
        extern char *optarg;
        extern int optind;
        FILE* output = stdout;

        int i = 0;
```

```
   printf("In tiff2psmain\n");

         while ((c = getopt(argc, argv, "h:w:d:o:O:aeps128DT")) != -1)

  printf("after while\n");
                 switch (c) {
                 case 'd':
                         dirnum = atoi(optarg);
                         break;
                 case 'D':
                         PSduplex = TRUE;
                         break;
                 case 'T':
                         PStumble = TRUE;
                         break;
                 case 'e':
                         generateEPSF = TRUE;
                         break;
                 case 'h':
                         pageHeight = atof(optarg);
                         break;
                 case 'o':
                         diroff = (uint32) strtoul(optarg, NULL, 0);
                         break;
                 case 'O':               /* XXX too bad -o is already taken */
                         output = fopen(optarg, "w");
 printf("opened\n");
                         if (output == NULL) {
                                 fprintf(stderr,
                                     "%s: %s: Cannot open output file.\n",
                                     argv[0], optarg);
                                 exit(-2);
                         }
                         break;
                 case 'a':
                         printAll = TRUE;
                         /* fall thru... */
                 case 'p':
                         generateEPSF = FALSE;
                         break;
                 case 's':
                         printAll = FALSE;
                         break;
                 case 'w':
                         pageWidth = atof(optarg);
                         break;
                 case '1':
                         level2 = FALSE;
                         ascii85 = FALSE;
                         break;
                 case '2':
                         level2 = TRUE;
                         ascii85 = TRUE;                 /* default to yes */
                         break;
                 case '8':
                         ascii85 = FALSE;
                         break;
                 case '?':
                         usage(-1);
                 }
         for (; argc - optind > 0; optind++) {
```

```
                TIFF* tif = TIFFOpen(filename = argv[optind], "r");
                if (tif != NULL) {
                        if (dirnum != -1 && !TIFFSetDirectory(tif, dirnum))
                                return (-1);
                        else if (diroff != 0 &&
                            !TIFFSetSubDirectory(tif, diroff))
                                return (-1);
                        np = TIFF2PS(output, tif, pageWidth, pageHeight);
                        TIFFClose(tif);
                }
        }
        if (np)
                PSTail(output, np);
        else
                usage(-1);
        if (output != stdout)
                fclose(output);
        return (0);
}
```

---

kfax'checkImage() (./kdegraphics/kfax/tiff2ps.c:224)

```
checkImage(TIFF* tif)
{
        switch (bitspersample) {
        case 1: case 2:
        case 4: case 8:
                break;
        default:
                TIFFError(filename, "Can not handle %d-bit/sample image",
                    bitspersample);
                return (0);
        }
        switch (photometric) {
        case PHOTOMETRIC_YCBCR:
                if (compression == COMPRESSION_JPEG &&
                    planarconfiguration == PLANARCONFIG_CONTIG) {
                        /* can rely on libjpeg to convert to RGB */
                        TIFFSetField(tif, TIFFTAG_JPEGCOLORMODE,
                                     JPEGCOLORMODE_RGB);
                        photometric = PHOTOMETRIC_RGB;
                } else {
                        if (level2)
                                break;
                        TIFFError(filename, "Can not handle image with %s",
                            "PhotometricInterpretation=YCbCr");
                        return (0);
                }
                /* fall thru... */
        case PHOTOMETRIC_RGB:
                if (alpha && bitspersample != 8) {
                        TIFFError(filename,
                            "Can not handle %d-bit/sample RGB image with alpha",
                            bitspersample);
                        return (0);
                }
                /* fall thru... */
```

```
        case PHOTOMETRIC_SEPARATED:
        case PHOTOMETRIC_PALETTE:
        case PHOTOMETRIC_MINISBLACK:
        case PHOTOMETRIC_MINISWHITE:
                break;
        case PHOTOMETRIC_CIELAB:
                /* fall thru... */
        default:
                TIFFError(filename,
                    "Can not handle image with PhotometricInterpretation=%d",
                    photometric);
                return (0);
        }
        if (planarconfiguration == PLANARCONFIG_SEPARATE && extrasamples > 0)
                TIFFWarning(filename, "Ignoring extra samples");
        return (1);
}
```

---

kfax'PhotoshopBanner() (./kdegraphics/kfax/tiff2ps.c:310)

```
PhotoshopBanner(FILE* fd, uint32 w, uint32 h, int bs, int nc, char* startline)
{
        fprintf(fd, "%%ImageData: %ld %ld %d %d 0 %d 2 \"",
            (long) w, (long) h, bitspersample, nc, bs);
        fprintf(fd, startline, nc);
        fprintf(fd, "\"\n");
}

static void
```

---

kfax'setupPageState() (./kdegraphics/kfax/tiff2ps.c:319)

```
setupPageState(TIFF* tif, uint32* pw, uint32* ph, float* pprw, float* pprh)
{
        uint16 res_unit;
        float xres, yres;

        TIFFGetField(tif, TIFFTAG_IMAGEWIDTH, pw);
        TIFFGetField(tif, TIFFTAG_IMAGELENGTH, ph);
        TIFFGetFieldDefaulted(tif, TIFFTAG_RESOLUTIONUNIT, &res_unit);
        /*
         * Calculate printable area.
         */
        if (!TIFFGetField(tif, TIFFTAG_XRESOLUTION, &xres))
                xres = PS_UNIT_SIZE;
        if (!TIFFGetField(tif, TIFFTAG_YRESOLUTION, &yres))
                yres = PS_UNIT_SIZE;
        switch (res_unit) {
        case RESUNIT_CENTIMETER:
                xres /= 2.54, yres /= 2.54;
                break;
        case RESUNIT_NONE:
                xres *= PS_UNIT_SIZE, yres *= PS_UNIT_SIZE;
                break;
        }
```

```
        *pprh = PSUNITS(*ph, yres);
        *pprw = PSUNITS(*pw, xres);
}

static int
```

---

kfax'isCCITTCompression() (./kdegraphics/kfax/tiff2ps.c:347)

```
isCCITTCompression(TIFF* tif)
{
    uint16 compress;
    TIFFGetField(tif, TIFFTAG_COMPRESSION, &compress);
    return (compress == COMPRESSION_CCITTFAX3 ||
            compress == COMPRESSION_CCITTFAX4 ||
            compress == COMPRESSION_CCITTRLE ||
            compress == COMPRESSION_CCITTRLEW);
}
```

---

kfax'TIFF2PS() (./kdegraphics/kfax/tiff2ps.c:366)

```
TIFF2PS(FILE* fd, TIFF* tif, float pw, float ph)
{
        uint32 w, h;
        float ox, oy, prw, prh;
        uint32 subfiletype;
        uint16* sampleinfo;
        static int npages = 0;

        if (!TIFFGetField(tif, TIFFTAG_XPOSITION, &ox))
                ox = 0;
        if (!TIFFGetField(tif, TIFFTAG_YPOSITION, &oy))
                oy = 0;
        setupPageState(tif, &w, &h, &prw, &prh);

        do {
                tf_numberstrips = TIFFNumberOfStrips(tif);
                TIFFGetFieldDefaulted(tif, TIFFTAG_ROWSPERSTRIP,
                    &tf_rowsperstrip);
                setupPageState(tif, &w, &h, &prw, &prh);
                if (!npages)
                        PSHead(fd, tif, w, h, prw, prh, ox, oy);
                tf_bytesperrow = TIFFScanlineSize(tif);
                TIFFGetFieldDefaulted(tif, TIFFTAG_BITSPERSAMPLE,
                    &bitspersample);
                TIFFGetFieldDefaulted(tif, TIFFTAG_SAMPLESPERPIXEL,
                    &samplesperpixel);
                TIFFGetFieldDefaulted(tif, TIFFTAG_PLANARCONFIG,
                    &planarconfiguration);
                TIFFGetField(tif, TIFFTAG_COMPRESSION, &compression);
                TIFFGetFieldDefaulted(tif, TIFFTAG_EXTRASAMPLES,
                    &extrasamples, &sampleinfo);
                alpha = (extrasamples == 1 &&
                         sampleinfo[0] == EXTRASAMPLE_ASSOCALPHA);
                if (!TIFFGetField(tif, TIFFTAG_PHOTOMETRIC, &photometric)) {
                        switch (samplesperpixel - extrasamples) {
```

```
                case 1:
                        if (isCCITTCompression(tif))
                                photometric = PHOTOMETRIC_MINISWHITE;
                        else
                                photometric = PHOTOMETRIC_MINISBLACK;
                        break;
                case 3:
                        photometric = PHOTOMETRIC_RGB;
                        break;
                }
        }
        if (checkImage(tif)) {
                npages++;
                fprintf(fd, "%%%%Page: %d %d\n", npages, npages);
                fprintf(fd, "gsave\n");
                fprintf(fd, "100 dict begin\n");
                if (pw != 0 && ph != 0)
                        fprintf(fd, "%f %f scale\n",
                            pw*PS_UNIT_SIZE, ph*PS_UNIT_SIZE);
                else
                        fprintf(fd, "%f %f scale\n", prw, prh);
                PSpage(fd, tif, w, h);
                fprintf(fd, "end\n");
                fprintf(fd, "grestore\n");
                fprintf(fd, "showpage\n");
        }
        if (generateEPSF)
                break;
        TIFFGetFieldDefaulted(tif, TIFFTAG_SUBFILETYPE, &subfiletype);
    } while (((subfiletype & FILETYPE_PAGE) || printAll) &&
        TIFFReadDirectory(tif));

    return(npages);
}
```

---

kfax'PSHead() (./kdegraphics/kfax/tiff2ps.c:460)

```
PSHead(FILE *fd, TIFF *tif, uint32 w, uint32 h, float pw, float ph,
        float ox, float oy)
{
        time_t t;

        (void) tif; (void) w; (void) h;
        t = time(0);
        fprintf(fd, "%%!PS-Adobe-3.0%s\n", generateEPSF ? " EPSF-3.0" : "");
        fprintf(fd, "%%%%Creator: kfax\n");
        fprintf(fd, "%%%%Title: %s\n", filename);
        fprintf(fd, "%%%%CreationDate: %s", ctime(&t));
        fprintf(fd, "%%%%DocumentData: Clean7Bit\n");
        fprintf(fd, "%%%%Origin: %ld %ld\n", (long) ox, (long) oy);
        /* NB: should use PageBoundingBox */
        fprintf(fd, "%%%%BoundingBox: 0 0 %ld %ld\n",
            (long) ceil(pw), (long) ceil(ph));
        fprintf(fd, "%%%%LanguageLevel: %d\n", level2 ? 2 : 1);
        fprintf(fd, "%%%%Pages: (atend)\n");
        fprintf(fd, "%%%%EndComments\n");
```

```
        fprintf(fd, "%%%%BeginSetup\n");
        if (PSduplex)
                fprintf(fd, "%s", DuplexPreamble);
        if (PStumble)
                fprintf(fd, "%s", TumblePreamble);
        fprintf(fd, "%%%%EndSetup\n");
}

void
```

---

kfax'PSTail() (./kdegraphics/kfax/tiff2ps.c:488)

```
PSTail(FILE *fd, int npages)
{
        fprintf(fd, "%%%%Trailer\n");
        fprintf(fd, "%%%%Pages: %d\n", npages);
        fprintf(fd, "%%%%EOF\n");
}

static int
```

---

kfax'checkcmap() (./kdegraphics/kfax/tiff2ps.c:496)

```
checkcmap(TIFF* tif, int n, uint16* r, uint16* g, uint16* b)
{
        (void) tif;
        while (n-- > 0)
                if (*r++ >= 256 || *g++ >= 256 || *b++ >= 256)
                        return (16);
        TIFFWarning(filename, "Assuming 8-bit colormap");
        return (8);
}

static void
```

---

kfax'PS_Lvl2colorspace() (./kdegraphics/kfax/tiff2ps.c:507)

```
PS_Lvl2colorspace(FILE* fd, TIFF* tif)
{
        uint16 *rmap, *gmap, *bmap;
        int i, num_colors;

        /*
         * Set up PostScript Level 2 colorspace according to
         * section 4.8 in the PostScript refenence manual.
         */
        fputs("% PostScript Level 2 only.\n", fd);
        if (photometric != PHOTOMETRIC_PALETTE) {
                if (photometric == PHOTOMETRIC_YCBCR) {
                    /* MORE CODE HERE */
                }
                fprintf(fd, "/Device%s",
                    samplesperpixel > 2 ? "RGB" : "Gray");
```

```
                fputs(" setcolorspace\n", fd);
                return;
        }

        /*
         * Set up an indexed/palette colorspace
         */
        num_colors = (1 << bitspersample);
        if (!TIFFGetField(tif, TIFFTAG_COLORMAP, &rmap, &gmap, &bmap)) {
                TIFFError(filename,
                        "Palette image w/o \"Colormap\" tag");
                return;
        }
        if (checkcmap(tif, num_colors, rmap, gmap, bmap) == 16) {
                /*
                 * Convert colormap to 8-bits values.
                 */
```

---

kfax'PS_Lvl2ImageDict() (./kdegraphics/kfax/tiff2ps.c:574)

```
PS_Lvl2ImageDict(FILE* fd, TIFF* tif, uint32 w, uint32 h)
{
        int use_rawdata;
        uint32 tile_width, tile_height;
        uint16 predictor, minsamplevalue, maxsamplevalue;
        int repeat_count;
        char im_h[64], im_x[64], im_y[64];

        (void)strcpy(im_x, "0");
        (void)sprintf(im_y, "%lu", (long) h);
        (void)sprintf(im_h, "%lu", (long) h);
        tile_width = w;
        tile_height = h;
        if (TIFFIsTiled(tif)) {
                repeat_count = TIFFNumberOfTiles(tif);
                TIFFGetField(tif, TIFFTAG_TILEWIDTH, &tile_width);
                TIFFGetField(tif, TIFFTAG_TILELENGTH, &tile_height);
                if (tile_width > w || tile_height > h ||
                    (w % tile_width) != 0 || (h % tile_height != 0)) {
                        /*
                         * The tiles does not fit image width and height.
                         * Set up a clip rectangle for the image unit square.
                         */
                        fputs("0 0 1 1 rectclip\n", fd);
                }
                if (tile_width < w) {
                        fputs("/im_x 0 def\n", fd);
                        (void)strcpy(im_x, "im_x neg");
                }
                if (tile_height < h) {
                        fputs("/im_y 0 def\n", fd);
                        (void)sprintf(im_y, "%lu im_y sub", (unsigned long) h);
                }
        } else {
                repeat_count = tf_numberstrips;
                tile_height = tf_rowsperstrip;
                if (tile_height > h)
                        tile_height = h;
```

```
        if (repeat_count > 1) {
                fputs("/im_y 0 def\n", fd);
                fprintf(fd, "/im_h %lu def\n",
                    (unsigned long) tile_height);
                (void)strcpy(im_h, "im_h");
                (void)sprintf(im_y, "%lu im_y sub", (unsigned long) h);
        }
}

/*
 * Output start of exec block
 */
fputs("{ % exec\n", fd);

if (repeat_count > 1)
        fprintf(fd, "%d { %% repeat\n", repeat_count);

/*
 * Output filter options and image dictionary.
 */
if (ascii85)
        fputs(" /im_stream currentfile /ASCII85Decode filter def\n",
            fd);
fputs(" <<\n", fd);
fputs("  /ImageType 1\n", fd);
fprintf(fd, "  /Width %lu\n", (unsigned long) tile_width);
fprintf(fd, "  /Height %lu\n", (unsigned long) tile_height);
if (planarconfiguration == PLANARCONFIG_SEPARATE)
        fputs("  /MultipleDataSources true\n", fd);
fprintf(fd, "  /ImageMatrix [ %lu 0 0 %ld %s %s ]\n",
    (unsigned long) w, - (long)h, im_x, im_y);
fprintf(fd, "  /BitsPerComponent %d\n", bitspersample);
fprintf(fd, "  /Interpolate %s\n", interpolate ? "true" : "false");

switch (samplesperpixel) {
case 1:
        switch (photometric) {
        case PHOTOMETRIC_MINISBLACK:
                fputs("  /Decode [0 1]\n", fd);
                break;
        case PHOTOMETRIC_MINISWHITE:
                switch (compression) {
                case COMPRESSION_CCITTRLE:
                case COMPRESSION_CCITTRLEW:
                case COMPRESSION_CCITTFAX3:
                case COMPRESSION_CCITTFAX4:
                        /*
                         * Manage inverting with /Blackis1 flag
                         * since there migth be uncompressed parts
                         */
                        fputs("  /Decode [0 1]\n", fd);
                        break;
                default:
                        /*
                         * ERROR...
                         */
                        fputs("  /Decode [1 0]\n", fd);
                        break;
                }
                break;
        case PHOTOMETRIC_PALETTE:
```

```
                TIFFGetFieldDefaulted(tif, TIFFTAG_MINSAMPLEVALUE,
                    &minsamplevalue);
                TIFFGetFieldDefaulted(tif, TIFFTAG_MAXSAMPLEVALUE,
                    &maxsamplevalue);
                fprintf(fd, "  /Decode [%u %u]\n",
                            minsamplevalue, maxsamplevalue);
                break;
        default:
                /*
                 * ERROR ?
                 */
                fputs("  /Decode [0 1]\n", fd);
                break;
        }
        break;
case 3:
        switch (photometric) {
        case PHOTOMETRIC_RGB:
                fputs("  /Decode [0 1 0 1 0 1]\n", fd);
                break;
        case PHOTOMETRIC_MINISWHITE:
        case PHOTOMETRIC_MINISBLACK:
        default:
                /*
                 * ERROR??
                 */
                fputs("  /Decode [0 1 0 1 0 1]\n", fd);
                break;
        }
        break;
case 4:
        /*
         * ERROR??
         */
        fputs("  /Decode [0 1 0 1 0 1 0 1]\n", fd);
        break;
}
fputs("  /DataSource", fd);
if (planarconfiguration == PLANARCONFIG_SEPARATE &&
    samplesperpixel > 1)
        fputs(" [", fd);
if (ascii85)
        fputs(" im_stream", fd);
else
        fputs(" currentfile /ASCIIHexDecode filter", fd);

use_rawdata = TRUE;
switch (compression) {
case COMPRESSION_NONE:  /* 1: uncompressed */
        break;
case COMPRESSION_CCITTRLE:      /* 2: CCITT modified Huffman RLE */
case COMPRESSION_CCITTRLEW:     /* 32771: #1 w/ word alignment */
case COMPRESSION_CCITTFAX3:     /* 3: CCITT Group 3 fax encoding */
case COMPRESSION_CCITTFAX4:     /* 4: CCITT Group 4 fax encoding */
        fputs("\n\t<<\n", fd);
        if (compression == COMPRESSION_CCITTFAX3) {
                uint32 g3_options;

                fputs("\t /EndOfLine true\n", fd);
                fputs("\t /EndOfBlock false\n", fd);
                if (!TIFFGetField(tif, TIFFTAG_GROUP3OPTIONS,
```

```
                                        &g3_options))
                        g3_options = 0;
                if (g3_options & GROUP3OPT_2DENCODING)
                        fprintf(fd, "\t /K %s\n", im_h);
                if (g3_options & GROUP3OPT_UNCOMPRESSED)
                        fputs("\t /Uncompressed true\n", fd);
                if (g3_options & GROUP3OPT_FILLBITS)
                        fputs("\t /EncodedByteAlign true\n", fd);
        }
        if (compression == COMPRESSION_CCITTFAX4) {
                uint32 g4_options;

                fputs("\t /K -1\n", fd);
                TIFFGetFieldDefaulted(tif, TIFFTAG_GROUP4OPTIONS,
                                       &g4_options);
                if (g4_options & GROUP4OPT_UNCOMPRESSED)
                        fputs("\t /Uncompressed true\n", fd);
        }
        if (!(tile_width == w && w == 1728U))
                fprintf(fd, "\t /Columns %lu\n",
                    (unsigned long) tile_width);
        fprintf(fd, "\t /Rows %s\n", im_h);
        if (compression == COMPRESSION_CCITTRLE ||
            compression == COMPRESSION_CCITTRLEW) {
                fputs("\t /EncodedByteAlign true\n", fd);
                fputs("\t /EndOfBlock false\n", fd);
        }
        if (photometric == PHOTOMETRIC_MINISBLACK)
                fputs("\t /BlackIs1 true\n", fd);
        fprintf(fd, "\t>> /CCITTFaxDecode filter");
        break;
    case COMPRESSION_LZW:   /* 5: Lempel-Ziv  & Welch */
        TIFFGetFieldDefaulted(tif, TIFFTAG_PREDICTOR, &predictor);
        if (predictor == 2) {
                fputs("\n\t<<\n", fd);
                fprintf(fd, "\t /Predictor %u\n", predictor);
                fprintf(fd, "\t /Columns %lu\n",
                    (unsigned long) tile_width);
                fprintf(fd, "\t /Colors %u\n", samplesperpixel);
                fprintf(fd, "\t /BitsPerComponent %u\n",
                    bitspersample);
                fputs("\t>>", fd);
        }
        fputs(" /LZWDecode filter", fd);
        break;
    case COMPRESSION_PACKBITS:              /* 32773: Macintosh RLE */
        fputs(" /RunLengthDecode filter", fd);
        use_rawdata = TRUE;
      break;
    case COMPRESSION_OJPEG: /* 6: !6.0 JPEG */
    case COMPRESSION_JPEG:  /* 7: %JPEG DCT compression */
#ifdef notdef
        /*
         * Code not tested yet
         */
        fputs(" /DCTDecode filter", fd);
        use_rawdata = TRUE;
#else
        use_rawdata = FALSE;
#endif
        break;
```

```
        case COMPRESSION_NEXT:          /* 32766: NeXT 2-bit RLE */
        case COMPRESSION_THUNDERSCAN:   /* 32809: ThunderScan RLE */
        case COMPRESSION_PIXARFILM:     /* 32908: Pixar companded 10bit LZW */
        case COMPRESSION_DEFLATE:       /* 32946: Deflate compression */
        case COMPRESSION_JBIG:          /* 34661: ISO JBIG */
                use_rawdata = FALSE;
                break;
        default:
                /*
                 * ERROR...
                 */
                use_rawdata = FALSE;
                break;
        }
        if (planarconfiguration == PLANARCONFIG_SEPARATE &&
            samplesperpixel > 1) {
                uint16 i;

                /*
                 * NOTE: This code does not work yet...
                 */
                for (i = 1; i < samplesperpixel; i++)
                        fputs(" dup", fd);
                fputs(" ]", fd);
        }
        fputs("\n >> image\n", fd);
        if (ascii85)
                fputs(" im_stream flushfile\n", fd);
        if (repeat_count > 1) {
                if (tile_width < w) {
                        fprintf(fd, " /im_x im_x %lu add def\n",
                            (unsigned long) tile_width);
                        if (tile_height < h) {
                                fprintf(fd, " im_x %lu ge {\n",
                                    (unsigned long) w);
                                fputs("  /im_x 0 def\n", fd);
                                fprintf(fd, " /im_y im_y %lu add def\n",
                                    (unsigned long) tile_height);
                                fputs(" } if\n", fd);
                        }
                }
                if (tile_height < h) {
                        if (tile_width >= w) {
                                fprintf(fd, " /im_y im_y %lu add def\n",
                                    (unsigned long) tile_height);
                                if (!TIFFIsTiled(tif)) {
                                        fprintf(fd, " /im_h %lu im_y sub",
                                            (unsigned long) h);
                                        fprintf(fd, " dup %lu gt { pop",
                                            (unsigned long) tile_height);
                                        fprintf(fd, " %lu } if def\n",
                                            (unsigned long) tile_height);
                                }
                        }
                }
                fputs("} repeat\n", fd);
        }
        /*
         * End of exec function
         */
        fputs("}\n", fd);
```

```
        return(use_rawdata);
}

int
```

---

kfax'PS_Lvl2page() (./kdegraphics/kfax/tiff2ps.c:861)

```
PS_Lvl2page(FILE* fd, TIFF* tif, uint32 w, uint32 h)
{
        uint16 fillorder;
        int use_rawdata, tiled_image, breaklen;
        uint32 chunk_no, num_chunks, *bc;
        unsigned char *buf_data, *cp;
        tsize_t chunk_size, byte_count;

        PS_Lvl2colorspace(fd, tif);
        use_rawdata = PS_Lvl2ImageDict(fd, tif, w, h);

        fputs("%%BeginData:\n", fd);
        fputs("exec\n", fd);

        tiled_image = TIFFIsTiled(tif);
        if (tiled_image) {
                num_chunks = TIFFNumberOfTiles(tif);
                TIFFGetField(tif, TIFFTAG_TILEBYTECOUNTS, &bc);
        } else {
                num_chunks = TIFFNumberOfStrips(tif);
                TIFFGetField(tif, TIFFTAG_STRIPBYTECOUNTS, &bc);
        }

        if (use_rawdata) {
                chunk_size = bc[0];
                for (chunk_no = 1; chunk_no < num_chunks; chunk_no++)
                        if (bc[chunk_no] > chunk_size)
                                chunk_size = bc[chunk_no];
        } else {
                if (tiled_image)
                        chunk_size = TIFFTileSize(tif);
                else
                        chunk_size = TIFFStripSize(tif);
        }
        buf_data = (unsigned char *)_TIFFmalloc(chunk_size);
        if (!buf_data) {
                TIFFError(filename, "Can't alloc %u bytes for %s.",
                        chunk_size, tiled_image ? "tiles" : "strips");
                return(FALSE);
        }

        TIFFGetFieldDefaulted(tif, TIFFTAG_FILLORDER, &fillorder);
        for (chunk_no = 0; chunk_no < num_chunks; chunk_no++) {
                if (ascii85)
                        Ascii85Init();
                else
                        breaklen = 36;
                if (use_rawdata) {
                        if (tiled_image)
                                byte_count = TIFFReadRawTile(tif, chunk_no,
```

```
                                    buf_data, chunk_size);
                else
                        byte_count = TIFFReadRawStrip(tif, chunk_no,
                                    buf_data, chunk_size);
                if (fillorder == FILLORDER_LSB2MSB)
                    TIFFReverseBits(buf_data, byte_count);
        } else {
                if (tiled_image)
                        byte_count = TIFFReadEncodedTile(tif,
                                        chunk_no, buf_data,
                                        chunk_size);
                else
                        byte_count = TIFFReadEncodedStrip(tif,
                                        chunk_no, buf_data,
                                        chunk_size);
        }
        if (byte_count < 0) {
                TIFFError(filename, "Can't read %s %d.",
                        tiled_image ? "tile" : "strip", chunk_no);
                if (ascii85)
                        Ascii85Put('\0', fd);
        }
        for (cp = buf_data; byte_count > 0; byte_count--) {
                if (ascii85)
                        Ascii85Put(*cp++, fd);
                else {
                        if (--breaklen <= 0) {
                                putc('\n', fd);
                                breaklen = 36;
                        }
                        putc(hex[((*cp)>>4)&0xf], fd);
                        putc(hex[(*cp)&0xf], fd);
                        cp++;
                }
        }
        if (ascii85)
                Ascii85Flush(fd);
        else
                putc('\n', fd);
    }
    _TIFFfree(buf_data);
    fputs("%%EndData\n", fd);
    return(TRUE);
}

void
```

---

kfax'PSpage() (./kdegraphics/kfax/tiff2ps.c:957)

```
PSpage(FILE* fd, TIFF* tif, uint32 w, uint32 h)
{
        if (level2 && PS_Lvl2page(fd, tif, w, h))
                return;
        ps_bytesperrow = tf_bytesperrow;
        switch (photometric) {
        case PHOTOMETRIC_RGB:
                if (planarconfiguration == PLANARCONFIG_CONTIG) {
                        fprintf(fd, "%s", RGBcolorimage);
```

```
                        PSColorContigPreamble(fd, w, h, 3);
                        PSDataColorContig(fd, tif, w, h, 3);
                } else {
                        PSColorSeparatePreamble(fd, w, h, 3);
                        PSDataColorSeparate(fd, tif, w, h, 3);
                }
                break;
        case PHOTOMETRIC_SEPARATED:
                /* XXX should emit CMYKcolorimage */
                if (planarconfiguration == PLANARCONFIG_CONTIG) {
                        PSColorContigPreamble(fd, w, h, 4);
                        PSDataColorContig(fd, tif, w, h, 4);
                } else {
                        PSColorSeparatePreamble(fd, w, h, 4);
                        PSDataColorSeparate(fd, tif, w, h, 4);
                }
                break;
        case PHOTOMETRIC_PALETTE:
                fprintf(fd, "%s", RGBcolorimage);
                PhotoshopBanner(fd, w, h, 1, 3, "false 3 colorimage");
                fprintf(fd, "/scanLine %ld string def\n",
                    (long) ps_bytesperrow);
                fprintf(fd, "%lu %lu 8\n",
                    (unsigned long) w, (unsigned long) h);
                fprintf(fd, "[%lu 0 0 -%lu 0 %lu]\n",
                    (unsigned long) w, (unsigned long) h, (unsigned long) h);
                fprintf(fd, "{currentfile scanLine readhexstring pop} bind\n");
                fprintf(fd, "false 3 colorimage\n");
                PSDataPalette(fd, tif, w, h);
                break;
        case PHOTOMETRIC_MINISBLACK:
        case PHOTOMETRIC_MINISWHITE:
                PhotoshopBanner(fd, w, h, 1, 1, "image");
                fprintf(fd, "/scanLine %ld string def\n",
                    (long) ps_bytesperrow);
                fprintf(fd, "%lu %lu %d\n",
                    (unsigned long) w, (unsigned long) h, bitspersample);
                fprintf(fd, "[%lu 0 0 -%lu 0 %lu]\n",
                    (unsigned long) w, (unsigned long) h, (unsigned long) h);
                fprintf(fd,
                    "{currentfile scanLine readhexstring pop} bind\n");
                fprintf(fd, "image\n");
                PSDataBW(fd, tif, w, h);
                break;
        }
        putc('\n', fd);
}

void
```

---

kfax'PSColorContigPreamble() (./kdegraphics/kfax/tiff2ps.c:1015)

```
PSColorContigPreamble(FILE* fd, uint32 w, uint32 h, int nc)
{
        ps_bytesperrow = nc * (tf_bytesperrow / samplesperpixel);
        PhotoshopBanner(fd, w, h, 1, nc, "false %d colorimage");
        fprintf(fd, "/line %ld string def\n", (long) ps_bytesperrow);
        fprintf(fd, "%lu %lu %d\n",
```

```
            (unsigned long) w, (unsigned long) h, bitspersample);
        fprintf(fd, "[%lu 0 0 -%lu 0 %lu]\n",
            (unsigned long) w, (unsigned long) h, (unsigned long) h);
        fprintf(fd, "{currentfile line readhexstring pop} bind\n");
        fprintf(fd, "false %d colorimage\n", nc);
}

void
```

---

kfax'PSColorSeparatePreamble() (./kdegraphics/kfax/tiff2ps.c:1029)

```
PSColorSeparatePreamble(FILE* fd, uint32 w, uint32 h, int nc)
{
        int i;

        PhotoshopBanner(fd, w, h, ps_bytesperrow, nc, "true %d colorimage");
        for (i = 0; i < nc; i++)
                fprintf(fd, "/line%d %ld string def\n",
                    i, (long) ps_bytesperrow);
        fprintf(fd, "%lu %lu %d\n",
            (unsigned long) w, (unsigned long) h, bitspersample);
        fprintf(fd, "[%lu 0 0 -%lu 0 %lu] \n",
            (unsigned long) w, (unsigned long) h, (unsigned long) h);
        for (i = 0; i < nc; i++)
                fprintf(fd, "{currentfile line%d readhexstring pop}bind\n", i);
        fprintf(fd, "true %d colorimage\n", nc);
}
```

---

kfax'PSDataColorContig() (./kdegraphics/kfax/tiff2ps.c:1055)

```
PSDataColorContig(FILE* fd, TIFF* tif, uint32 w, uint32 h, int nc)
{
        uint32 row;
        int breaklen = MAXLINE, cc, es = samplesperpixel - nc;
        unsigned char *tf_buf;
        unsigned char *cp, c;

        (void) w;
        tf_buf = (unsigned char *) _TIFFmalloc(tf_bytesperrow);
        if (tf_buf == NULL) {
                TIFFError(filename, "No space for scanline buffer");
                return;
        }
        for (row = 0; row < h; row++) {
                if (TIFFReadScanline(tif, tf_buf, row, 0) < 0)
                        break;
                cp = tf_buf;
                if (alpha) {
                        int adjust;
                        cc = 0;
                        for (; cc < tf_bytesperrow; cc += samplesperpixel) {
                                DOBREAK(breaklen, nc, fd);
                                /*
                                 * For images with alpha, matte against
                                 * a white background; i.e.
```

```
                                 *    Cback * (1 - Aimage)
                                 * where Cback = 1.
                                 */
                                adjust = 255 - cp[nc];
                                switch (nc) {
                                case 4: c = *cp++ + adjust; PUTHEX(c,fd);
                                case 3: c = *cp++ + adjust; PUTHEX(c,fd);
                                case 2: c = *cp++ + adjust; PUTHEX(c,fd);
                                case 1: c = *cp++ + adjust; PUTHEX(c,fd);
                                }
                                cp += es;
                        }
                } else {
                        cc = 0;
                        for (; cc < tf_bytesperrow; cc += samplesperpixel) {
                                DOBREAK(breaklen, nc, fd);
                                switch (nc) {
                                case 4: c = *cp++; PUTHEX(c,fd);
                                case 3: c = *cp++; PUTHEX(c,fd);
                                case 2: c = *cp++; PUTHEX(c,fd);
                                case 1: c = *cp++; PUTHEX(c,fd);
                                }
                                cp += es;
                        }
                }
        }
        _TIFFfree((char *) tf_buf);
}

void
```

---

kfax'PSDataColorSeparate() (./kdegraphics/kfax/tiff2ps.c:1110)

```
PSDataColorSeparate(FILE* fd, TIFF* tif, uint32 w, uint32 h, int nc)
{
        uint32 row;
        int breaklen = MAXLINE, cc, s, maxs;
        unsigned char *tf_buf;
        unsigned char *cp, c;

        (void) w;
        tf_buf = (unsigned char *) _TIFFmalloc(tf_bytesperrow);
        if (tf_buf == NULL) {
                TIFFError(filename, "No space for scanline buffer");
                return;
        }
        maxs = (samplesperpixel > nc ? nc : samplesperpixel);
        for (row = 0; row < h; row++) {
                for (s = 0; s < maxs; s++) {
                        if (TIFFReadScanline(tif, tf_buf, row, s) < 0)
                                break;
                        for (cp = tf_buf, cc = 0; cc < tf_bytesperrow; cc++) {
                                DOBREAK(breaklen, 1, fd);
                                c = *cp++;
                                PUTHEX(c,fd);
                        }
                }
        }
```

```
        _TIFFfree((char *) tf_buf);
}
```

---

kfax'PSDataPalette() (./kdegraphics/kfax/tiff2ps.c:1142)

```
PSDataPalette(FILE* fd, TIFF* tif, uint32 w, uint32 h)
{
        uint16 *rmap, *gmap, *bmap;
        uint32 row;
        int breaklen = MAXLINE, cc, nc;
        unsigned char *tf_buf;
        unsigned char *cp, c;

        (void) w;
        if (!TIFFGetField(tif, TIFFTAG_COLORMAP, &rmap, &gmap, &bmap)) {
                TIFFError(filename, "Palette image w/o \"Colormap\" tag");
                return;
        }
        switch (bitspersample) {
        case 8: case 4: case 2: case 1:
                break;
        default:
                TIFFError(filename, "Depth %d not supported", bitspersample);
                return;
        }
        nc = 3 * (8 / bitspersample);
        tf_buf = (unsigned char *) _TIFFmalloc(tf_bytesperrow);
        if (tf_buf == NULL) {
                TIFFError(filename, "No space for scanline buffer");
                return;
        }
        if (checkcmap(tif, 1<<bitspersample, rmap, gmap, bmap) == 16) {
                int i;
```

---

kfax'PSDataBW() (./kdegraphics/kfax/tiff2ps.c:1214)

```
PSDataBW(FILE* fd, TIFF* tif, uint32 w, uint32 h)
{
        int breaklen = MAXLINE;
        unsigned char* tf_buf;
        unsigned char* cp;
        tsize_t stripsize = TIFFStripSize(tif);
        tstrip_t s;

        (void) w; (void) h;
        tf_buf = (unsigned char *) _TIFFmalloc(stripsize);
        if (tf_buf == NULL) {
                TIFFError(filename, "No space for scanline buffer");
                return;
        }
        if (ascii85)
                Ascii85Init();
        for (s = 0; s < TIFFNumberOfStrips(tif); s++) {
                int cc = TIFFReadEncodedStrip(tif, s, tf_buf, stripsize);
                if (cc < 0) {
```

```
                                TIFFError(filename, "Can't read strip");
                                break;
                        }
                        cp = tf_buf;
                        if (photometric == PHOTOMETRIC_MINISWHITE) {
                                for (cp += cc; --cp >= tf_buf;)
                                        *cp = ~*cp;
                                cp++;
                        }
                        if (ascii85) {
                                while (cc-- > 0)
                                        Ascii85Put(*cp++, fd);
                        } else {
                                while (cc-- > 0) {
                                        unsigned char c = *cp++;
                                        DOBREAK(breaklen, 1, fd);
                                        PUTHEX(c, fd);
                                }
                        }
                }
                if (ascii85)
                        Ascii85Flush(fd);
                else if (level2)
                        fputs(">\n", fd);
                _TIFFfree(tf_buf);
}

void
```

---

kfax'PSRawDataBW() (./kdegraphics/kfax/tiff2ps.c:1261)

```
PSRawDataBW(FILE* fd, TIFF* tif, uint32 w, uint32 h)
{
        uint32 *bc;
        uint32 bufsize;
        int breaklen = MAXLINE, cc;
        uint16 fillorder;
        unsigned char *tf_buf;
        unsigned char *cp, c;
        tstrip_t s;

        (void) w; (void) h;
        TIFFGetFieldDefaulted(tif, TIFFTAG_FILLORDER, &fillorder);
        TIFFGetField(tif, TIFFTAG_STRIPBYTECOUNTS, &bc);
        bufsize = bc[0];
        tf_buf = (unsigned char*) _TIFFmalloc(bufsize);
        if (tf_buf == NULL) {
                TIFFError(filename, "No space for strip buffer");
                return;
        }
        for (s = 0; s < tf_numberstrips; s++) {
                if (bc[s] > bufsize) {
                        tf_buf = (unsigned char *) _TIFFrealloc(tf_buf, bc[s]);
                        if (tf_buf == NULL) {
                                TIFFError(filename,
                                    "No space for strip buffer");
                                return;
                        }
```

```
                        bufsize = bc[s];
                }
                cc = TIFFReadRawStrip(tif, s, tf_buf, bc[s]);
                if (cc < 0) {
                        TIFFError(filename, "Can't read strip");
                        break;
                }
                if (fillorder == FILLORDER_LSB2MSB)
                        TIFFReverseBits(tf_buf, cc);
                if (!ascii85) {
                        for (cp = tf_buf; cc > 0; cc--) {
                                DOBREAK(breaklen, 1, fd);
                                c = *cp++;
                                PUTHEX(c, fd);
                        }
                        fputs(">\n", fd);
                        breaklen = MAXLINE;
                } else {
                        Ascii85Init();
                        for (cp = tf_buf; cc > 0; cc--)
                                Ascii85Put(*cp++, fd);
                        Ascii85Flush(fd);
                }
        }
        _TIFFfree((char *) tf_buf);
}

void
```

---

kfax'Ascii85Init() (./kdegraphics/kfax/tiff2ps.c:1316)

```
Ascii85Init(void)
{
        ascii85breaklen = 2*MAXLINE;
        ascii85count = 0;
}

static char*
```

---

kfax'Ascii85Encode() (./kdegraphics/kfax/tiff2ps.c:1323)

```
Ascii85Encode(unsigned char* raw)
{
        static char encoded[6];
        uint32 word;

        word = (((raw[0]<<8)+raw[1])<<16) + (raw[2]<<8) + raw[3];
        if (word != 0L) {
                uint32 q;
                uint16 w1;

                q = word / (85L*85*85*85);      /* actually only a byte */
                encoded[0] = q + '!';

                word -= q * (85L*85*85*85); q = word / (85L*85*85);
                encoded[1] = q + '!';
```

```
                word -= q * (85L*85*85); q = word / (85*85);
                encoded[2] = q + '!';

                w1 = (uint16) (word - q*(85L*85));
                encoded[3] = (w1 / 85) + '!';
                encoded[4] = (w1 % 85) + '!';
                encoded[5] = '\0';
        } else
                encoded[0] = 'z', encoded[1] = '\0';
        return (encoded);
}

void
```

---

kfax'Ascii85Put() (./kdegraphics/kfax/tiff2ps.c:1352)

```
Ascii85Put(unsigned char code, FILE* fd)
{
        ascii85buf[ascii85count++] = code;
        if (ascii85count >= 4) {
                unsigned char* p;
                int n;

                for (n = ascii85count, p = ascii85buf; n >= 4; n -= 4, p += 4) {
                        char* cp;
                        for (cp = Ascii85Encode(p); *cp; cp++) {
                                putc(*cp, fd);
                                if (--ascii85breaklen == 0) {
                                        putc('\n', fd);
                                        ascii85breaklen = 2*MAXLINE;
                                }
                        }
                        p += 4;
                }
                _TIFFmemcpy(ascii85buf, p, n);
                ascii85count = n;
        }
}

void
```

---

kfax'Ascii85Flush() (./kdegraphics/kfax/tiff2ps.c:1376)

```
Ascii85Flush(FILE* fd)
{
        if (ascii85count > 0) {
                char* res;
                _TIFFmemset(&ascii85buf[ascii85count], 0, 3);
                res = Ascii85Encode(ascii85buf);
                fwrite(res[0] == 'z' ? "!!!!" : res, ascii85count + 1, 1, fd);
        }
        fputs("~>\n", fd);
}
```

---

kfax'usage() (./kdegraphics/kfax/tiff2ps.c:1408)

```
usage(int code)
{

        char buf[BUFSIZ];
        int i;

  return;


        setbuf(stderr, buf);
        for (i = 0; tiff2psstuff[i] != NULL; i++)
                fprintf(stderr, "%s\n", tiff2psstuff[i]);
        exit(code);
}
```

---

kfax'viewfax_addCmdLineOptions() (./kdegraphics/kfax/viewfax.cpp:188)

```
viewfax_addCmdLineOptions()
{
  KCmdLineArgs::addCmdLineOptions( options );
}
```

---

kfax'viewfaxmain() (./kdegraphics/kfax/viewfax.cpp:193)

```
int viewfaxmain()
{
    int banner = 0;
    int have_height = 0;

    bo.i = 1;
    defaultpage.vres = -1;
    have_no_fax = TRUE;

    /* TODO Do I need to know this: */
    defaultpage.expander = g31expand;

    ProgName = "KFax";

    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

    if (args->isSet("height"))
    {
       have_height = 1;
       defaultpage.height = args->getOption("height").toInt();
    }

    if (args->isSet("2"))
    {
       defaultpage.expander = g32expand;
```

```
   if(!have_height)
      defaultpage.height = 2339;
}

if (args->isSet("4"))
{
   defaultpage.expander = g4expand;
   if(!have_height)
      defaultpage.height = 2155;
}

if (args->isSet("invert"))
{
   defaultpage.inverse = 1;
}

if (args->isSet("landscape"))
{
   defaultpage.orient |= TURN_L;
}

if (args->isSet("fine"))
{
   defaultpage.vres = 1;
}

if (!args->isSet("rmal")) // "normal" is interpreted as "no"+"rmal" :-)
{
   defaultpage.vres = 0;
}

if (args->isSet("reverse"))
{
   defaultpage.lsbfirst = 1;
}

if (args->isSet("upsidedown"))
{
   defaultpage.orient |= TURN_U;
}

if (args->isSet("width"))
{
   defaultpage.width = args->getOption("width").toInt();
}

QCString mem = args->getOption("mem");
Memlimit = atoi(mem.data());
switch(mem[mem.length()-1]) {
        case 'M':
        case 'm':
            Memlimit *= 1024;
        case 'K':
        case 'k':
            Memlimit *= 1024;
}

if (defaultpage.expander == g4expand && defaultpage.height == 0) {
    KCmdLineArgs::usage("--height value is required to interpret raw g4 faxe
}
```

```
    firstpage = lastpage = thispage = helppage = auxpage =  0;

    for (int i = 0; i < args->count(); i++){
        (void) notetiff(QFile::decodeName(args->arg(i)));
    }
    args->clear();

    if (banner ) {
      fprintf(stderr, Banner);
      exit(1);
    }

    if ( firstpage == 0) {
      have_no_fax = TRUE;
    }
    else{
      have_no_fax = FALSE;
    }

    Disp = qtdisplay;
    Default_Screen = XDefaultScreen(qtdisplay);

    return 1;
}



/* return mismatching suffix of option name */
/*static char *suffix(char *opt, const char *prefix){

    while (*opt && *opt == *prefix) {
        opt++; prefix++;
    }
    return opt;
}
*/

/* Change orientation of all following pages */
```

---

kfax'TurnFollowing() (./kdegraphics/kfax/viewfax.cpp:317)

```
void TurnFollowing(int How, struct pagenode *pn)
{
    while (pn) {
        if (Pimage(pn)) {
            FreeImage(Pimage(pn));
            pn->extra = 0;
        }
        pn->orient ^= How;
        pn = pn->next;
    }
}

static void
```

---

kfax'drawline() (./kdegraphics/kfax/viewfax.cpp:330)

```
drawline(pixnum *run, int LineNum, struct pagenode *pn)
{
    t32bits *p, *p1;            /* p - current line, p1 - low-res duplicate */
    pixnum *r;                  /* pointer to run-lengths */
    t32bits pix;                /* current pixel value */
    t32bits acc;                /* pixel accumulator */
    int nacc;                   /* number of valid bits in acc */
    int tot;                    /* total pixels in line */
    int n;

    LineNum += pn->stripnum * pn->rowsperstrip;
    p = (t32bits *) (Pimage(pn)->data + LineNum*(2-pn->vres)*Pimage(pn)->bytes_pe
    p1 =(t32bits *)( pn->vres ? 0 : p + Pimage(pn)->bytes_per_line/sizeof(*p));
    r = run;
    acc = 0;
    nacc = 0;
    pix = pn->inverse ? ~0 : 0;
    tot = 0;
    while (tot < pn->width) {
        n = *r++;
        tot += n;
        if (pix)
            acc |= (~(t32bits)0 >> nacc);
        else if (nacc)
            acc &= (~(t32bits)0 << (32 - nacc));
        else
            acc = 0;
        if (nacc + n < 32) {
            nacc += n;
            pix = ~pix;
            continue;
        }
        *p++ = acc;
        if (p1)
            *p1++ = acc;
        n -= 32 - nacc;
        while (n >= 32) {
            n -= 32;
            *p++ = pix;
            if (p1)
                *p1++ = pix;
        }
        acc = pix;
        nacc = n;
        pix = ~pix;
    }
    if (nacc) {
        *p++ = acc;
        if (p1)
            *p1++ = acc;
    }
}

static int
```

---

kfax'GetPartImage() (./kdegraphics/kfax/viewfax.cpp:384)

```
GetPartImage(struct pagenode *pn, int n)
{
    unsigned char *Data = getstrip(pn, n);

    if (Data == 0)
        return 3;
    pn->stripnum = n;
    (*pn->expander)(pn, drawline);
    free(Data);
    return 1;
}
```

---

kfax'GetImage() (./kdegraphics/kfax/viewfax.cpp:396)

```
int GetImage(struct pagenode *pn){
    int i;

    if (pn->strips == 0) {

      /*printf("RAW fax file\n");*/

        /* raw file; maybe we don't have the height yet */
        unsigned char *Data = getstrip(pn, 0);
        if (Data == 0){

          return 0;
        }
        pn->extra = NewImage(pn->width, pn->vres ?
                             pn->height : 2*pn->height, 0, 1);

//printf("height = %d\n",pn->height);
//printf("setting height to %d\n", pn->vres ? pn->height : 2*pn->height);

        if(pn->extra == 0)
          return 0;

        (*pn->expander)(pn, drawline);
    }
    else {
        /* multi-strip tiff */
      /*printf("MULTI STRIP TIFF fax file\n");*/

        pn->extra = NewImage(pn->width, pn->vres ?
                             pn->height : 2*pn->height, 0, 1);

        if(pn->extra == 0)
          return 0;
        pn->stripnum = 0;
        for (i = 0; i < pn->nstrips; i++){

          int k =GetPartImage(pn, i);

          if ( k == 3 ){
            FreeImage(Pimage(pn));
            return k;
          }
```

```
        }
    }
    if (pn->orient & TURN_U)
        pn->extra = FlipImage(Pimage(pn));
    if (pn->orient & TURN_M)
        pn->extra = MirrorImage(Pimage(pn));
    if (pn->orient & TURN_L)
        pn->extra = RotImage(Pimage(pn));
    if (verbose) printf("\tmemused = %d\n", Memused);

/*
if(pn->extra)
  printf("pn->extra !=0 %s\n",pn->name);
else
  printf("pn->extra ==0 %s\n",pn->name);
  */

    return 1;
}



/* run this region through perl to generate the zoom table:
$lim = 1;
@c = ("0", "1", "1", "2");
print "static unsigned char Z[] = {\n";
for ($i = 0; $i < 16; $i++) {
    for ($j = 0; $j < 16; $j++) {
        $b1 = ($c[$j&3]+$c[$i&3]) > $lim;
        $b2 = ($c[($j>>2)&3]+$c[($i>>2)&3]) > $lim;
        printf " %X,", ($b2 << 1) | $b1;
    }
    print "\n";
}
print "};\n";
*/
```

---

kfax'ZoomImage() (./kdegraphics/kfax/viewfax.cpp:503)

```
XImage *ZoomImage(XImage *Big){

    XImage *Small;
    int w, h;
    int i, j;

    XDefineCursor(Disp, Win, WorkCursor);
    XFlush(Disp);
    w = (Big->width+1) / 2;
    h = (Big->height+1) / 2;
    Small = NewImage(w, h, 0, Big->bitmap_bit_order);
    if(Small == 0)
      return 0;

    Small->xoffset = (Big->xoffset+1)/2;
    for (i = 0; i < Big->height; i += 2) {
        t32bits *pb0 = (t32bits *) (Big->data + i * Big->bytes_per_line);
```

```
        t32bits *pb1 = pb0 + ((i == Big->height-1) ? 0 : Big->bytes_per_line/4);
        t32bits *ps = (t32bits *) (Small->data + i * Small->bytes_per_line / 2);
        for (j = 0; j < Big->bytes_per_line/8; j++) {
            t32bits r1, r2;
            t32bits t0 = *pb0++;
            t32bits t1 = *pb1++;
            r1 = (zak(nib(7,t0),nib(7,t1))<<14) |
                 (zak(nib(6,t0),nib(6,t1))<<12) |
                 (zak(nib(5,t0),nib(5,t1))<<10) |
                 (zak(nib(4,t0),nib(4,t1))<<8) |
                 (zak(nib(3,t0),nib(3,t1))<<6) |
                 (zak(nib(2,t0),nib(2,t1))<<4) |
                 (zak(nib(1,t0),nib(1,t1))<<2) |
                 (zak(nib(0,t0),nib(0,t1)));
            t0 = *pb0++;
            t1 = *pb1++;
            r2 = (zak(nib(7,t0),nib(7,t1))<<14) |
                 (zak(nib(6,t0),nib(6,t1))<<12) |
                 (zak(nib(5,t0),nib(5,t1))<<10) |
                 (zak(nib(4,t0),nib(4,t1))<<8) |
                 (zak(nib(3,t0),nib(3,t1))<<6) |
                 (zak(nib(2,t0),nib(2,t1))<<4) |
                 (zak(nib(1,t0),nib(1,t1))<<2) |
                 (zak(nib(0,t0),nib(0,t1)));
            *ps++ = (Big->bitmap_bit_order) ?
                (r1<<16)|r2 : (r2<<16)|r1;
        }
        for ( ; j < Small->bytes_per_line/4; j++) {
            t32bits r1;
            t32bits t0 = *pb0++;
            t32bits t1 = *pb1++;
            r1 = (zak(nib(7,t0),nib(7,t1))<<14) |
                 (zak(nib(6,t0),nib(6,t1))<<12) |
                 (zak(nib(5,t0),nib(5,t1))<<10) |
                 (zak(nib(4,t0),nib(4,t1))<<8) |
                 (zak(nib(3,t0),nib(3,t1))<<6) |
                 (zak(nib(2,t0),nib(2,t1))<<4) |
                 (zak(nib(1,t0),nib(1,t1))<<2) |
                 (zak(nib(0,t0),nib(0,t1)));
            *ps++ = (Big->bitmap_bit_order) ?
                (r1<<16) : r1;
        }
    }
    XDefineCursor(Disp, Win, ReadyCursor);
    return Small;
}
```

---

kfax'FlipImage() (./kdegraphics/kfax/viewfax.cpp:567)

```
XImage *FlipImage(XImage *Image){

    XImage *New = NewImage(Image->width, Image->height,
                           Image->data, !Image->bitmap_bit_order);
    if(New == 0)
      return 0;

    t32bits *p1 = (t32bits *) Image->data;
```

```
    t32bits *p2 = (t32bits *) (Image->data + Image->height *
                               Image->bytes_per_line - 4);

    /* the first shall be last ... */
    while (p1 < p2) {
        t32bits t = *p1;
        *p1++ = *p2;
        *p2-- = t;
    }

    /* let Xlib twiddle the bits */
    New->xoffset = 32 - (Image->width & 31) - Image->xoffset;
    New->xoffset &= 31;

    Image->data = 0;
    FreeImage(Image);
    return New;
}
```

---

kfax'MirrorImage() (./kdegraphics/kfax/viewfax.cpp:594)

```
XImage *MirrorImage(XImage *Image){

    int i;
    XImage *New = NewImage(Image->width, Image->height,
                           Image->data, !Image->bitmap_bit_order);
    if(New == 0)
      return 0;

    /* reverse order of 32-bit words in each line */
    for (i = 0; i < Image->height; i++) {
        t32bits *p1 = (t32bits *) (Image->data + Image->bytes_per_line * i);
        t32bits *p2 = p1 + Image->bytes_per_line/4 - 1;
        while (p1 < p2) {
            t32bits t = *p1;
            *p1++ = *p2;
            *p2-- = t;
        }
    }

    /* let Xlib twiddle the bits */
    New->xoffset = 32 - (Image->width & 31) - Image->xoffset;
    New->xoffset &= 31;

    Image->data = 0;
    FreeImage(Image);
    return New;
}
```

---

kfax'RotImage() (./kdegraphics/kfax/viewfax.cpp:622)

```
XImage *RotImage(XImage *Image){

    XImage *New;
```

```
    int w = Image->height;
    int h = Image->width;
    int i, j, k, shift;
    int order = Image->bitmap_bit_order;
    int offs = h+Image->xoffset-1;

    New = NewImage(w, h, 0, 1);
    if (New == 0)
      return 0;

    k = (32 - Image->xoffset) & 3;
    for (i = h - 1; i && k; i--, k--) {
        t32bits *sp = (t32bits *) Image->data + (offs-i)/32;
        t32bits *dp = (t32bits *) (New->data+i*New->bytes_per_line);
        t32bits d0 =0;
        shift = (offs-i)&31;
        if (order) shift = 31-shift;
        for (j = 0; j < w; j++) {
            t32bits t = *sp;
            sp += Image->bytes_per_line/4;
            d0 |= ((t >> shift) & 1);
            if ((j & 31) == 31)
                *dp++ = d0;
            d0 <<= 1;;
        }
        if (j & 31)
            *dp++ = d0<<(31-j);
    }
    for ( ; i >= 3; i-=4) {
        t32bits *sp = (t32bits *) Image->data + (offs-i)/32;
        t32bits *dp0 = (t32bits *) (New->data+i*New->bytes_per_line);
        t32bits *dp1 = dp0 - New->bytes_per_line/4;
        t32bits *dp2 = dp1 - New->bytes_per_line/4;
        t32bits *dp3 = dp2 - New->bytes_per_line/4;
        t32bits d0=0 , d1=0, d2 =0, d3 =0;
        shift = (offs-i)&31;
        if (order) shift = 28-shift;
        for (j = 0; j < w; j++) {
            t32bits t = *sp >> shift;
            sp += Image->bytes_per_line/4;
            d0 |= t & 1; t >>= 1;
            d1 |= t & 1; t >>= 1;
            d2 |= t & 1; t >>= 1;
            d3 |= t & 1; t >>= 1;
            if ((j & 31) == 31) {
                if (order) {
                    *dp0++ = d3;
                    *dp1++ = d2;
                    *dp2++ = d1;
                    *dp3++ = d0;
                }
                else {
                    *dp0++ = d0;
                    *dp1++ = d1;
                    *dp2++ = d2;
                    *dp3++ = d3;
                }
            }
            d0 <<= 1; d1 <<= 1; d2 <<= 1; d3 <<= 1;
        }
        if (j & 31) {
```

```
            if (order) {
                *dp0++ = d3<<(31-j);
                *dp1++ = d2<<(31-j);
                *dp2++ = d1<<(31-j);
                *dp3++ = d0<<(31-j);
            }
            else {
                *dp0++ = d0<<(31-j);
                *dp1++ = d1<<(31-j);
                *dp2++ = d2<<(31-j);
                *dp3++ = d3<<(31-j);
            }
        }
    }
    for (; i >= 0; i--) {
        t32bits *sp = (t32bits *) Image->data + (offs-i)/32;
        t32bits *dp = (t32bits *) (New->data+i*New->bytes_per_line);
        t32bits d0=0;
        shift = (offs-i)&31;
        if (order) shift = 31-shift;
        for (j = 0; j < w; j++) {
            t32bits t = *sp;
            sp += Image->bytes_per_line/4;
            d0 |= ((t >> shift) & 1);
            if ((j & 31) == 31)
                *dp++ = d0;
            d0 <<= 1;;
        }
        if (j & 31)
            *dp++ = d0<<(31-j);
    }
    FreeImage(Image);
    return New;
}

/* release some non-essential memory or abort */
```

---

kfax'release() (./kdegraphics/kfax/viewfax.cpp:730)

```
release(int quit)
{
  (void) quit;

    struct pagenode *pn;

    if (thispage) {
        /* first consider "uninteresting" pages */
        for (pn = firstpage->next; pn; pn = pn->next)
            if (pn->extra && pn != thispage && pn != thispage->prev &&
                pn != thispage->next && pn != lastpage) {
                FreeImage(Pimage(pn));
                pn->extra = 0;
                return 1;
            }
        Try(lastpage);
        Try(firstpage);
        Try(thispage->prev);
        Try(thispage->next);
```

```
    }

    return 0;

}
```

---

kfax'NewImage() (./kdegraphics/kfax/viewfax.cpp:755)

```
XImage *NewImage(int w, int h, char *data, int bit_order){

    XImage *newimage;
    /* This idea is taken from xwud/xpr.  Use a fake display with the
       desired bit/byte order to get the image routines initialised
       correctly */
    Display fake;

    fake = *Disp;
    if (data == 0)
        data = xmalloc(((w + 31) & ~31) * h / 8);
    fake.byte_order = ByteOrder;
    fake.bitmap_bit_order = bit_order;

    int returncode = -1;
    while ((newimage = XCreateImage(&fake, DefaultVisual(Disp, Default_Screen),
                                1, XYBitmap, 0, data, w, h, 32, 0)) == 0 ){

      returncode = release(1);
      if (returncode == 0)
        break;
    }

    if (returncode == 0){
      kfaxerror("Sorry","Can not allocate Memory for a new Fax Image\n");
      return 0;
    }

    Memused += newimage->bytes_per_line * newimage->height;
    /*printf("allocating %d bytes for %ld\n",
       newimage->bytes_per_line * newimage->height,
       newimage);*/


    return newimage;
}
```

---

kfax'FreeImage() (./kdegraphics/kfax/viewfax.cpp:792)

```
void FreeImage(XImage *Image){

  if (Image->data){
        Memused -= Image->bytes_per_line * Image->height;
/*printf("deallocating %d bytes for %ld\n",
       Image->bytes_per_line * Image->height,
       Image);*/
```

```
  }
    XDestroyImage(Image);
    setstatusbarmem(Memused);

}

#ifndef xmalloc
char *
```

---

kfax'xmalloc() (./kdegraphics/kfax/viewfax.cpp:807)

```
xmalloc(unsigned int size)
{
    char *p;

    while (Memused + size > Memlimit && release(0))
        ;
    while ((p = (char*) malloc(size)) == 0)
        (void) release(1);
    return p;
}
```