/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* XVID MPEG-4 VIDEO CODEC

\* - Encoder main module -

\*

\* Copyright(C) 2002-2010 Michael Militzer <isibaar@xvid.org>

\* 2002-2003 Peter Ross <pross@xvid.org>

\* 2002 Daniel Smith <danielsmith@astroboymail.com>

\*

\* This program is free software ; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation ; either version 2 of the License, or

\* (at your option) any later version.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY ; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with this program ; if not, write to the Free Software

\* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

\*

\* $Id: encoder.c 1985 2011-05-18 09:02:35Z Isibaar $

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <string.h>

#include "encoder.h"

#include "prediction/mbprediction.h"

#include "global.h"

#include "utils/timer.h"

#include "image/image.h"

#include "image/font.h"

#include "motion/sad.h"

#include "motion/motion.h"

#include "motion/gmc.h"

#include "bitstream/cbp.h"

#include "utils/mbfunctions.h"

#include "bitstream/bitstream.h"

#include "bitstream/mbcoding.h"

#include "utils/emms.h"

#include "bitstream/mbcoding.h"

#include "quant/quant\_matrix.h"

#include "utils/mem\_align.h"

# include "motion/motion\_smp.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Local function prototypes

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static int FrameCodeI(Encoder \* pEnc,

Bitstream \* bs);

static int FrameCodeP(Encoder \* pEnc,

Bitstream \* bs);

static void FrameCodeB(Encoder \* pEnc,

FRAMEINFO \* frame,

Bitstream \* bs);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Encoder creation

\*

\* This function creates an Encoder instance, it allocates all necessary

\* image buffers (reference, current and bframes) and initialize the internal

\* xvid encoder paremeters according to the XVID\_ENC\_PARAM input parameter.

\*

\* The code seems to be very long but is very basic, mainly memory allocation

\* and cleaning code.

\*

\* Returned values :

\* - 0 - no errors

\* - XVID\_ERR\_MEMORY - the libc could not allocate memory, the function

\* cleans the structure before exiting.

\* pParam->handle is also set to NULL.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* Simplify the "fincr/fbase" fraction

\*/

static int

gcd(int a, int b)

{

int r ;

if (b > a) {

r = a;

a = b;

b = r;

}

while ((r = a % b)) {

a = b;

b = r;

}

return b;

}

static void

simplify\_time(int \*inc, int \*base)

{

/\* common factor \*/

const int s = gcd(\*inc, \*base);

\*inc /= s;

\*base /= s;

if (\*base > 65535 || \*inc > 65535) {

int \*biggest;

int \*other;

float div;

if (\*base > \*inc) {

biggest = base;

other = inc;

} else {

biggest = inc;

other = base;

}

div = ((float)\*biggest)/((float)65535);

\*biggest = (unsigned int)(((float)\*biggest)/div);

\*other = (unsigned int)(((float)\*other)/div);

}

}

int

enc\_create(xvid\_enc\_create\_t \* create)

{

Encoder \*pEnc;

int n;

if (XVID\_VERSION\_MAJOR(create->version) != 1) /\* v1.x.x \*/

return XVID\_ERR\_VERSION;

if (create->width%2 || create->height%2)

return XVID\_ERR\_FAIL;

if (create->width<=0 || create->height<=0)

return XVID\_ERR\_FAIL;

/\* allocate encoder struct \*/

pEnc = (Encoder \*) xvid\_malloc(sizeof(Encoder), CACHE\_LINE);

if (pEnc == NULL)

return XVID\_ERR\_MEMORY;

memset(pEnc, 0, sizeof(Encoder));

pEnc->mbParam.profile = create->profile;

/\* global flags \*/

pEnc->mbParam.global\_flags = create->global;

if ((pEnc->mbParam.global\_flags & XVID\_GLOBAL\_PACKED))

pEnc->mbParam.global\_flags |= XVID\_GLOBAL\_DIVX5\_USERDATA;

/\* width, height \*/

pEnc->mbParam.width = create->width;

pEnc->mbParam.height = create->height;

pEnc->mbParam.mb\_width = (pEnc->mbParam.width + 15) / 16;

pEnc->mbParam.mb\_height = (pEnc->mbParam.height + 15) / 16;

pEnc->mbParam.edged\_width = 16 \* pEnc->mbParam.mb\_width + 2 \* EDGE\_SIZE;

pEnc->mbParam.edged\_height = 16 \* pEnc->mbParam.mb\_height + 2 \* EDGE\_SIZE;

/\* framerate \*/

pEnc->mbParam.fincr = MAX(create->fincr, 0);

pEnc->mbParam.fbase = create->fincr <= 0 ? 25 : create->fbase;

if (pEnc->mbParam.fincr>0)

simplify\_time((int\*)&pEnc->mbParam.fincr, (int\*)&pEnc->mbParam.fbase);

/\* zones \*/

if(create->num\_zones > 0) {

pEnc->num\_zones = create->num\_zones;

pEnc->zones = xvid\_malloc(sizeof(xvid\_enc\_zone\_t) \* pEnc->num\_zones, CACHE\_LINE);

if (pEnc->zones == NULL)

goto xvid\_err\_memory0;

memcpy(pEnc->zones, create->zones, sizeof(xvid\_enc\_zone\_t) \* pEnc->num\_zones);

} else {

pEnc->num\_zones = 0;

pEnc->zones = NULL;

}

/\* plugins \*/

if(create->num\_plugins > 0) {

pEnc->num\_plugins = create->num\_plugins;

pEnc->plugins = xvid\_malloc(sizeof(xvid\_enc\_plugin\_t) \* pEnc->num\_plugins, CACHE\_LINE);

if (pEnc->plugins == NULL)

goto xvid\_err\_memory0;

} else {

pEnc->num\_plugins = 0;

pEnc->plugins = NULL;

}

for (n=0; n<pEnc->num\_plugins;n++) {

xvid\_plg\_create\_t pcreate;

xvid\_plg\_info\_t pinfo;

memset(&pinfo, 0, sizeof(xvid\_plg\_info\_t));

pinfo.version = XVID\_VERSION;

if (create->plugins[n].func(NULL, XVID\_PLG\_INFO, &pinfo, NULL) >= 0) {

pEnc->mbParam.plugin\_flags |= pinfo.flags;

}

memset(&pcreate, 0, sizeof(xvid\_plg\_create\_t));

pcreate.version = XVID\_VERSION;

pcreate.num\_zones = pEnc->num\_zones;

pcreate.zones = pEnc->zones;

pcreate.width = pEnc->mbParam.width;

pcreate.height = pEnc->mbParam.height;

pcreate.mb\_width = pEnc->mbParam.mb\_width;

pcreate.mb\_height = pEnc->mbParam.mb\_height;

pcreate.fincr = pEnc->mbParam.fincr;

pcreate.fbase = pEnc->mbParam.fbase;

pcreate.param = create->plugins[n].param;

pEnc->plugins[n].func = NULL; /\* disable plugins that fail \*/

if (create->plugins[n].func(NULL, XVID\_PLG\_CREATE, &pcreate, &pEnc->plugins[n].param) >= 0) {

pEnc->plugins[n].func = create->plugins[n].func;

}

}

if ((pEnc->mbParam.global\_flags & XVID\_GLOBAL\_EXTRASTATS\_ENABLE) ||

(pEnc->mbParam.plugin\_flags & XVID\_REQPSNR)) {

pEnc->mbParam.plugin\_flags |= XVID\_REQORIGINAL; /\* psnr calculation requires the original \*/

}

/\* temp dquants \*/

if ((pEnc->mbParam.plugin\_flags & XVID\_REQDQUANTS)) {

pEnc->temp\_dquants = (int \*) xvid\_malloc(pEnc->mbParam.mb\_width \*

pEnc->mbParam.mb\_height \* sizeof(int), CACHE\_LINE);

if (pEnc->temp\_dquants==NULL)

goto xvid\_err\_memory1a;

}

/\* temp lambdas \*/

if (pEnc->mbParam.plugin\_flags & XVID\_REQLAMBDA) {

pEnc->temp\_lambda = (float \*) xvid\_malloc(pEnc->mbParam.mb\_width \*

pEnc->mbParam.mb\_height \* 6 \* sizeof(float), CACHE\_LINE);

if (pEnc->temp\_lambda == NULL)

goto xvid\_err\_memory1a;

}

/\* bframes \*/

pEnc->mbParam.max\_bframes = MAX(create->max\_bframes, 0);

pEnc->mbParam.bquant\_ratio = MAX(create->bquant\_ratio, 0);

pEnc->mbParam.bquant\_offset = create->bquant\_offset;

/\* min/max quant \*/

for (n=0; n<3; n++) {

pEnc->mbParam.min\_quant[n] = create->min\_quant[n] > 0 ? create->min\_quant[n] : 2;

pEnc->mbParam.max\_quant[n] = create->max\_quant[n] > 0 ? create->max\_quant[n] : 31;

}

/\* frame drop ratio \*/

pEnc->mbParam.frame\_drop\_ratio = MAX(create->frame\_drop\_ratio, 0);

/\* max keyframe interval \*/

pEnc->mbParam.iMaxKeyInterval = create->max\_key\_interval <= 0 ? (10 \* (int)pEnc->mbParam.fbase) / (int)pEnc->mbParam.fincr : create->max\_key\_interval;

/\* allocate working frame-image memory \*/

pEnc->current = xvid\_malloc(sizeof(FRAMEINFO), CACHE\_LINE);

pEnc->reference = xvid\_malloc(sizeof(FRAMEINFO), CACHE\_LINE);

if (pEnc->current == NULL || pEnc->reference == NULL)

goto xvid\_err\_memory1;

/\* allocate macroblock memory \*/

pEnc->current->mbs =

xvid\_malloc(sizeof(MACROBLOCK) \* pEnc->mbParam.mb\_width \*

pEnc->mbParam.mb\_height, CACHE\_LINE);

pEnc->reference->mbs =

xvid\_malloc(sizeof(MACROBLOCK) \* pEnc->mbParam.mb\_width \*

pEnc->mbParam.mb\_height, CACHE\_LINE);

if (pEnc->current->mbs == NULL || pEnc->reference->mbs == NULL)

goto xvid\_err\_memory2;

/\* allocate quant matrix memory \*/

pEnc->mbParam.mpeg\_quant\_matrices =

xvid\_malloc(sizeof(uint16\_t) \* 64 \* 8, CACHE\_LINE);

if (pEnc->mbParam.mpeg\_quant\_matrices == NULL)

goto xvid\_err\_memory2a;

/\* allocate interpolation image memory \*/

if ((pEnc->mbParam.plugin\_flags & XVID\_REQORIGINAL)) {

image\_null(&pEnc->sOriginal);

image\_null(&pEnc->sOriginal2);

}

image\_null(&pEnc->f\_refh);

image\_null(&pEnc->f\_refv);

image\_null(&pEnc->f\_refhv);

image\_null(&pEnc->current->image);

image\_null(&pEnc->reference->image);

image\_null(&pEnc->vInterH);

image\_null(&pEnc->vInterV);

image\_null(&pEnc->vInterHV);

if ((pEnc->mbParam.plugin\_flags & XVID\_REQORIGINAL)) {

if (image\_create

(&pEnc->sOriginal, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

if (image\_create

(&pEnc->sOriginal2, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

}

if (image\_create

(&pEnc->f\_refh, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

if (image\_create

(&pEnc->f\_refv, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

if (image\_create

(&pEnc->f\_refhv, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

if (image\_create

(&pEnc->current->image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

if (image\_create

(&pEnc->reference->image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

if (image\_create

(&pEnc->vInterH, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

if (image\_create

(&pEnc->vInterV, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

if (image\_create

(&pEnc->vInterHV, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

/\* Create full bitplane for GMC, this might be wasteful \*/

if (image\_create

(&pEnc->vGMC, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory3;

/\* init bframe image buffers \*/

pEnc->bframenum\_head = 0;

pEnc->bframenum\_tail = 0;

pEnc->flush\_bframes = 0;

pEnc->closed\_bframenum = -1;

/\* B Frames specific init \*/

pEnc->bframes = NULL;

if (pEnc->mbParam.max\_bframes > 0) {

pEnc->bframes =

xvid\_malloc(pEnc->mbParam.max\_bframes \* sizeof(FRAMEINFO \*),

CACHE\_LINE);

if (pEnc->bframes == NULL)

goto xvid\_err\_memory3;

for (n = 0; n < pEnc->mbParam.max\_bframes; n++)

pEnc->bframes[n] = NULL;

for (n = 0; n < pEnc->mbParam.max\_bframes; n++) {

pEnc->bframes[n] = xvid\_malloc(sizeof(FRAMEINFO), CACHE\_LINE);

if (pEnc->bframes[n] == NULL)

goto xvid\_err\_memory4;

pEnc->bframes[n]->mbs =

xvid\_malloc(sizeof(MACROBLOCK) \* pEnc->mbParam.mb\_width \*

pEnc->mbParam.mb\_height, CACHE\_LINE);

if (pEnc->bframes[n]->mbs == NULL)

goto xvid\_err\_memory4;

image\_null(&pEnc->bframes[n]->image);

if (image\_create

(&pEnc->bframes[n]->image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory4;

}

}

/\* init incoming frame queue \*/

pEnc->queue\_head = 0;

pEnc->queue\_tail = 0;

pEnc->queue\_size = 0;

pEnc->queue =

xvid\_malloc((pEnc->mbParam.max\_bframes+1) \* sizeof(QUEUEINFO),

CACHE\_LINE);

if (pEnc->queue == NULL)

goto xvid\_err\_memory4;

for (n = 0; n < pEnc->mbParam.max\_bframes+1; n++)

image\_null(&pEnc->queue[n].image);

for (n = 0; n < pEnc->mbParam.max\_bframes+1; n++) {

if (image\_create

(&pEnc->queue[n].image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height) < 0)

goto xvid\_err\_memory5;

}

/\* timestamp stuff \*/

pEnc->mbParam.m\_stamp = 0;

pEnc->m\_framenum = create->start\_frame\_num;

pEnc->current->stamp = 0;

pEnc->reference->stamp = 0;

/\* other stuff \*/

pEnc->iFrameNum = 0;

pEnc->fMvPrevSigma = -1;

/\* slices \*/

pEnc->num\_slices = MIN(MAX(1, create->num\_slices), (int) pEnc->mbParam.mb\_height);

/\* multithreaded stuff \*/

if (create->num\_threads > 0) {

#ifndef HAVE\_PTHREAD

int t = MAX(1, create->num\_threads);

#else

int t = MIN(create->num\_threads, (int) (pEnc->mbParam.mb\_height>>1)); /\* at least two rows per thread \*/

#endif

int threads\_per\_slice = MAX(1, (t / pEnc->num\_slices));

int rows\_per\_thread = (pEnc->mbParam.mb\_height + threads\_per\_slice - 1) / threads\_per\_slice;

pEnc->num\_threads = t;

pEnc->smpData = xvid\_malloc(t\*sizeof(SMPData), CACHE\_LINE);

if (!pEnc->smpData)

goto xvid\_err\_nosmp;

/\* tmp bitstream buffer for slice coding \*/

pEnc->smpData[0].tmp\_buffer = xvid\_malloc(16\*pEnc->mbParam.edged\_width\*pEnc->mbParam.mb\_height\*sizeof(uint8\_t), CACHE\_LINE);

if (! pEnc->smpData[0].tmp\_buffer) goto xvid\_err\_nosmp;

for (n = 0; n < t; n++) {

int s = MIN(pEnc->num\_threads, pEnc->num\_slices);

pEnc->smpData[n].complete\_count\_self =

xvid\_malloc(rows\_per\_thread \* sizeof(int), CACHE\_LINE);

if (!pEnc->smpData[n].complete\_count\_self)

goto xvid\_err\_nosmp;

if (n > 0 && n < s) {

pEnc->smpData[n].bs = (Bitstream \*) xvid\_malloc(sizeof(Bitstream), CACHE\_LINE);

if (!pEnc->smpData[n].bs)

goto xvid\_err\_nosmp;

pEnc->smpData[n].sStat = (Statistics \*) xvid\_malloc(sizeof(Statistics), CACHE\_LINE);

if (!pEnc->smpData[n].sStat)

goto xvid\_err\_nosmp;

pEnc->smpData[n].tmp\_buffer = pEnc->smpData[0].tmp\_buffer + 16\*(((n-1)\*pEnc->mbParam.edged\_width\*pEnc->mbParam.mb\_height)/s);

BitstreamInit(pEnc->smpData[n].bs, pEnc->smpData[n].tmp\_buffer, 0);

}

if (n != 0)

pEnc->smpData[n].complete\_count\_above =

pEnc->smpData[n-1].complete\_count\_self;

}

pEnc->smpData[0].complete\_count\_above =

pEnc->smpData[t-1].complete\_count\_self - 1;

} else {

xvid\_err\_nosmp:

/\* no SMP \*/

if (pEnc->smpData) {

if (pEnc->smpData[0].tmp\_buffer)

xvid\_free(pEnc->smpData[0].tmp\_buffer);

}

else {

pEnc->smpData = xvid\_malloc(1\*sizeof(SMPData), CACHE\_LINE);

if (pEnc->smpData == NULL)

goto xvid\_err\_memory5;

}

create->num\_threads = 0;

}

create->handle = (void \*) pEnc;

init\_timer();

init\_mpeg\_matrix(pEnc->mbParam.mpeg\_quant\_matrices);

return 0; /\* ok \*/

/\*

\* We handle all XVID\_ERR\_MEMORY here, this makes the code lighter

\*/

xvid\_err\_memory5:

for (n = 0; n < pEnc->mbParam.max\_bframes+1; n++) {

image\_destroy(&pEnc->queue[n].image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

}

xvid\_free(pEnc->queue);

xvid\_err\_memory4:

if (pEnc->mbParam.max\_bframes > 0) {

int i;

for (i = 0; i < pEnc->mbParam.max\_bframes; i++) {

if (pEnc->bframes[i] == NULL)

continue;

image\_destroy(&pEnc->bframes[i]->image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

xvid\_free(pEnc->bframes[i]->mbs);

xvid\_free(pEnc->bframes[i]);

}

xvid\_free(pEnc->bframes);

}

xvid\_err\_memory3:

if ((pEnc->mbParam.plugin\_flags & XVID\_REQORIGINAL)) {

image\_destroy(&pEnc->sOriginal, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->sOriginal2, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

}

image\_destroy(&pEnc->f\_refh, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->f\_refv, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->f\_refhv, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->current->image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->reference->image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->vInterH, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->vInterV, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->vInterHV, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

/\* destroy GMC image \*/

image\_destroy(&pEnc->vGMC, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

xvid\_err\_memory2a:

xvid\_free(pEnc->mbParam.mpeg\_quant\_matrices);

xvid\_err\_memory2:

xvid\_free(pEnc->current->mbs);

xvid\_free(pEnc->reference->mbs);

xvid\_err\_memory1:

xvid\_free(pEnc->current);

xvid\_free(pEnc->reference);

xvid\_err\_memory1a:

if ((pEnc->mbParam.plugin\_flags & XVID\_REQDQUANTS)) {

xvid\_free(pEnc->temp\_dquants);

}

if(pEnc->mbParam.plugin\_flags & XVID\_REQLAMBDA) {

xvid\_free(pEnc->temp\_lambda);

}

xvid\_err\_memory0:

for (n=0; n<pEnc->num\_plugins;n++) {

if (pEnc->plugins[n].func) {

pEnc->plugins[n].func(pEnc->plugins[n].param, XVID\_PLG\_DESTROY, NULL, NULL);

}

}

xvid\_free(pEnc->plugins);

xvid\_free(pEnc->zones);

xvid\_free(pEnc);

create->handle = NULL;

return XVID\_ERR\_MEMORY;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Encoder destruction

\*

\* This function destroy the entire encoder structure created by a previous

\* successful enc\_create call.

\*

\* Returned values (for now only one returned value) :

\* - 0 - no errors

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int

enc\_destroy(Encoder \* pEnc)

{

int i;

/\* B Frames specific \*/

for (i = 0; i < pEnc->mbParam.max\_bframes+1; i++) {

image\_destroy(&pEnc->queue[i].image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

}

xvid\_free(pEnc->queue);

if (pEnc->mbParam.max\_bframes > 0) {

for (i = 0; i < pEnc->mbParam.max\_bframes; i++) {

if (pEnc->bframes[i] == NULL)

continue;

image\_destroy(&pEnc->bframes[i]->image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

xvid\_free(pEnc->bframes[i]->mbs);

xvid\_free(pEnc->bframes[i]);

}

xvid\_free(pEnc->bframes);

}

/\* All images, reference, current etc ... \*/

image\_destroy(&pEnc->current->image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->reference->image, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->vInterH, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->vInterV, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->vInterHV, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->f\_refh, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->f\_refv, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->f\_refhv, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->vGMC, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

if ((pEnc->mbParam.plugin\_flags & XVID\_REQORIGINAL)) {

image\_destroy(&pEnc->sOriginal, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

image\_destroy(&pEnc->sOriginal2, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height);

}

/\* Encoder structure \*/

xvid\_free(pEnc->current->mbs);

xvid\_free(pEnc->current);

xvid\_free(pEnc->reference->mbs);

xvid\_free(pEnc->reference);

if ((pEnc->mbParam.plugin\_flags & XVID\_REQDQUANTS)) {

xvid\_free(pEnc->temp\_dquants);

}

if ((pEnc->mbParam.plugin\_flags & XVID\_REQLAMBDA)) {

xvid\_free(pEnc->temp\_lambda);

}

if (pEnc->num\_plugins>0) {

xvid\_plg\_destroy\_t pdestroy;

memset(&pdestroy, 0, sizeof(xvid\_plg\_destroy\_t));

pdestroy.version = XVID\_VERSION;

pdestroy.num\_frames = pEnc->m\_framenum;

for (i=0; i<pEnc->num\_plugins;i++) {

if (pEnc->plugins[i].func) {

pEnc->plugins[i].func(pEnc->plugins[i].param, XVID\_PLG\_DESTROY, &pdestroy, NULL);

}

}

xvid\_free(pEnc->plugins);

}

xvid\_free(pEnc->mbParam.mpeg\_quant\_matrices);

if (pEnc->num\_zones > 0)

xvid\_free(pEnc->zones);

if (pEnc->num\_threads > 0) {

for (i = 1; i < MAX(1, MIN(pEnc->num\_threads, pEnc->num\_slices)); i++) {

xvid\_free(pEnc->smpData[i].bs);

xvid\_free(pEnc->smpData[i].sStat);

}

if (pEnc->smpData[0].tmp\_buffer) xvid\_free(pEnc->smpData[0].tmp\_buffer);

for (i = 0; i < pEnc->num\_threads; i++)

xvid\_free(pEnc->smpData[i].complete\_count\_self);

}

xvid\_free(pEnc->smpData);

xvid\_free(pEnc);

return 0; /\* ok \*/

}

/\*

call the plugins

\*/

static void call\_plugins(Encoder \* pEnc, FRAMEINFO \* frame, IMAGE \* original,

int opt, int \* type, int \* quant, xvid\_enc\_stats\_t \* stats)

{

unsigned int i, j, k;

xvid\_plg\_data\_t data;

/\* set data struct \*/

memset(&data, 0, sizeof(xvid\_plg\_data\_t));

data.version = XVID\_VERSION;

/\* find zone \*/

for(i=0; i<pEnc->num\_zones && pEnc->zones[i].frame<=frame->frame\_num; i++) ;

data.zone = i>0 ? &pEnc->zones[i-1] : NULL;

data.width = pEnc->mbParam.width;

data.height = pEnc->mbParam.height;

data.mb\_width = pEnc->mbParam.mb\_width;

data.mb\_height = pEnc->mbParam.mb\_height;

data.fincr = frame->fincr;

data.fbase = pEnc->mbParam.fbase;

data.bquant\_ratio = pEnc->mbParam.bquant\_ratio;

data.bquant\_offset = pEnc->mbParam.bquant\_offset;

for (i=0; i<3; i++) {

data.min\_quant[i] = pEnc->mbParam.min\_quant[i];

data.max\_quant[i] = pEnc->mbParam.max\_quant[i];

}

data.reference.csp = XVID\_CSP\_PLANAR;

data.reference.plane[0] = pEnc->reference->image.y;

data.reference.plane[1] = pEnc->reference->image.u;

data.reference.plane[2] = pEnc->reference->image.v;

data.reference.stride[0] = pEnc->mbParam.edged\_width;

data.reference.stride[1] = pEnc->mbParam.edged\_width/2;

data.reference.stride[2] = pEnc->mbParam.edged\_width/2;

data.current.csp = XVID\_CSP\_PLANAR;

data.current.plane[0] = frame->image.y;

data.current.plane[1] = frame->image.u;

data.current.plane[2] = frame->image.v;

data.current.stride[0] = pEnc->mbParam.edged\_width;

data.current.stride[1] = pEnc->mbParam.edged\_width/2;

data.current.stride[2] = pEnc->mbParam.edged\_width/2;

data.frame\_num = frame->frame\_num;

if (opt == XVID\_PLG\_BEFORE) {

data.type = \*type;

data.quant = \*quant;

data.vol\_flags = frame->vol\_flags;

data.vop\_flags = frame->vop\_flags;

data.motion\_flags = frame->motion\_flags;

} else if (opt == XVID\_PLG\_FRAME) {

data.type = coding2type(frame->coding\_type);

data.quant = frame->quant;

if ((pEnc->mbParam.plugin\_flags & XVID\_REQDQUANTS)) {

data.dquant = pEnc->temp\_dquants;

data.dquant\_stride = pEnc->mbParam.mb\_width;

memset(data.dquant, 0, data.mb\_width\*data.mb\_height\*sizeof(int));

}

if(pEnc->mbParam.plugin\_flags & XVID\_REQLAMBDA) {

int block = 0;

emms();

data.lambda = pEnc->temp\_lambda;

for(i = 0;i < pEnc->mbParam.mb\_height; i++)

for(j = 0;j < pEnc->mbParam.mb\_width; j++)

for (k = 0; k < 6; k++)

data.lambda[block++] = 1.0f;

}

} else { /\* XVID\_PLG\_AFTER \*/

if ((pEnc->mbParam.plugin\_flags & XVID\_REQORIGINAL)) {

data.original.csp = XVID\_CSP\_PLANAR;

data.original.plane[0] = original->y;

data.original.plane[1] = original->u;

data.original.plane[2] = original->v;

data.original.stride[0] = pEnc->mbParam.edged\_width;

data.original.stride[1] = pEnc->mbParam.edged\_width/2;

data.original.stride[2] = pEnc->mbParam.edged\_width/2;

}

if ((frame->vol\_flags & XVID\_VOL\_EXTRASTATS) ||

(pEnc->mbParam.plugin\_flags & XVID\_REQPSNR)) {

data.sse\_y =

plane\_sse( original->y, frame->image.y,

pEnc->mbParam.edged\_width, pEnc->mbParam.width,

pEnc->mbParam.height);

data.sse\_u =

plane\_sse( original->u, frame->image.u,

pEnc->mbParam.edged\_width/2, pEnc->mbParam.width/2,

pEnc->mbParam.height/2);

data.sse\_v =

plane\_sse( original->v, frame->image.v,

pEnc->mbParam.edged\_width/2, pEnc->mbParam.width/2,

pEnc->mbParam.height/2);

}

data.type = coding2type(frame->coding\_type);

data.quant = frame->quant;

if ((pEnc->mbParam.plugin\_flags & XVID\_REQDQUANTS)) {

data.dquant = pEnc->temp\_dquants;

data.dquant\_stride = pEnc->mbParam.mb\_width;

for (j=0; j<pEnc->mbParam.mb\_height; j++)

for (i=0; i<pEnc->mbParam.mb\_width; i++) {

data.dquant[j\*data.dquant\_stride + i] = frame->mbs[j\*pEnc->mbParam.mb\_width + i].dquant;

}

}

data.vol\_flags = frame->vol\_flags;

data.vop\_flags = frame->vop\_flags;

data.motion\_flags = frame->motion\_flags;

data.length = frame->length;

data.kblks = frame->sStat.kblks;

data.mblks = frame->sStat.mblks;

data.ublks = frame->sStat.ublks;

/\* New code \*/

data.stats.type = coding2type(frame->coding\_type);

data.stats.quant = frame->quant;

data.stats.vol\_flags = frame->vol\_flags;

data.stats.vop\_flags = frame->vop\_flags;

data.stats.length = frame->length;

data.stats.hlength = frame->length - (frame->sStat.iTextBits / 8);

data.stats.kblks = frame->sStat.kblks;

data.stats.mblks = frame->sStat.mblks;

data.stats.ublks = frame->sStat.ublks;

data.stats.sse\_y = data.sse\_y;

data.stats.sse\_u = data.sse\_u;

data.stats.sse\_v = data.sse\_v;

if (stats)

\*stats = data.stats;

}

/\* call plugins \*/

for (i=0; i<(unsigned int)pEnc->num\_plugins;i++) {

emms();

if (pEnc->plugins[i].func) {

if (pEnc->plugins[i].func(pEnc->plugins[i].param, opt, &data, NULL) < 0) {

continue;

}

}

}

emms();

/\* copy modified values back into frame\*/

if (opt == XVID\_PLG\_BEFORE) {

\*type = data.type;

\*quant = data.quant > 0 ? data.quant : 2; /\* default \*/

frame->vol\_flags = data.vol\_flags;

frame->vop\_flags = data.vop\_flags;

frame->motion\_flags = data.motion\_flags;

} else if (opt == XVID\_PLG\_FRAME) {

if ((pEnc->mbParam.plugin\_flags & XVID\_REQDQUANTS)) {

for (j=0; j<pEnc->mbParam.mb\_height; j++)

for (i=0; i<pEnc->mbParam.mb\_width; i++) {

frame->mbs[j\*pEnc->mbParam.mb\_width + i].dquant = data.dquant[j\*data.mb\_width + i];

}

} else {

for (j=0; j<pEnc->mbParam.mb\_height; j++)

for (i=0; i<pEnc->mbParam.mb\_width; i++) {

frame->mbs[j\*pEnc->mbParam.mb\_width + i].dquant = 0;

}

}

if (pEnc->mbParam.plugin\_flags & XVID\_REQLAMBDA) {

for (j = 0; j < pEnc->mbParam.mb\_height; j++)

for (i = 0; i < pEnc->mbParam.mb\_width; i++)

for (k = 0; k < 6; k++) {

frame->mbs[j\*pEnc->mbParam.mb\_width + i].lambda[k] =

(int) ((float)(1<<LAMBDA\_EXP) \* data.lambda[6 \* (j \* data.mb\_width + i) + k]);

}

} else {

for (j = 0; j<pEnc->mbParam.mb\_height; j++)

for (i = 0; i<pEnc->mbParam.mb\_width; i++)

for (k = 0; k < 6; k++) {

frame->mbs[j\*pEnc->mbParam.mb\_width + i].lambda[k] = 1<<LAMBDA\_EXP;

}

}

frame->mbs[0].quant = data.quant; /\* FRAME will not affect the quant in stats \*/

}

}

static \_\_inline void inc\_frame\_num(Encoder \* pEnc)

{

pEnc->current->frame\_num = pEnc->m\_framenum;

pEnc->current->stamp = pEnc->mbParam.m\_stamp; /\* first frame is zero \*/

pEnc->mbParam.m\_stamp += pEnc->current->fincr;

pEnc->m\_framenum++; /\* debug ticker \*/

}

static \_\_inline void dec\_frame\_num(Encoder \* pEnc)

{

pEnc->mbParam.m\_stamp -= pEnc->mbParam.fincr;

pEnc->m\_framenum--; /\* debug ticker \*/

}

static \_\_inline void

MBSetDquant(MACROBLOCK \* pMB, int x, int y, MBParam \* mbParam)

{

if (pMB->cbp == 0) {

/\* we want to code dquant but the quantizer value will not be used yet

let's find out if we can postpone dquant to next MB

\*/

if (x == mbParam->mb\_width-1 && y == mbParam->mb\_height-1) {

pMB->dquant = 0; /\* it's the last MB of all, the easiest case \*/

return;

} else {

MACROBLOCK \* next = pMB + 1;

const MACROBLOCK \* prev = pMB - 1;

if (next->mode != MODE\_INTER4V && next->mode != MODE\_NOT\_CODED)

/\* mode allows dquant change in the future \*/

if (abs(next->quant - prev->quant) <= 2) {

/\* quant change is not out of range \*/

pMB->quant = prev->quant;

pMB->dquant = 0;

next->dquant = next->quant - prev->quant;

return;

}

}

}

/\* couldn't skip this dquant \*/

pMB->mode = MODE\_INTER\_Q;

}

static \_\_inline void

set\_timecodes(FRAMEINFO\* pCur,FRAMEINFO \*pRef, int32\_t time\_base)

{

pCur->ticks = (int32\_t)pCur->stamp % time\_base;

pCur->seconds = ((int32\_t)pCur->stamp / time\_base) - ((int32\_t)pRef->stamp / time\_base) ;

#if 0 /\* HEAVY DEBUG OUTPUT \*/

fprintf(stderr,"WriteVop: %d - %d \n",

((int32\_t)pCur->stamp / time\_base), ((int32\_t)pRef->stamp / time\_base));

fprintf(stderr,"set\_timecodes: VOP %1d stamp=%lld ref\_stamp=%lld base=%d\n",

pCur->coding\_type, pCur->stamp, pRef->stamp, time\_base);

fprintf(stderr,"set\_timecodes: VOP %1d seconds=%d ticks=%d (ref-sec=%d ref-tick=%d)\n",

pCur->coding\_type, pCur->seconds, pCur->ticks, pRef->seconds, pRef->ticks);

#endif

}

static void

simplify\_par(int \*par\_width, int \*par\_height)

{

int \_par\_width = (!\*par\_width) ? 1 : (\*par\_width<0) ? -\*par\_width: \*par\_width;

int \_par\_height = (!\*par\_height) ? 1 : (\*par\_height<0) ? -\*par\_height: \*par\_height;

int divisor = gcd(\_par\_width, \_par\_height);

\_par\_width /= divisor;

\_par\_height /= divisor;

/\* 2^8 precision maximum \*/

if (\_par\_width>255 || \_par\_height>255) {

float div;

emms();

if (\_par\_width>\_par\_height)

div = (float)\_par\_width/255;

else

div = (float)\_par\_height/255;

\_par\_width = (int)((float)\_par\_width/div);

\_par\_height = (int)((float)\_par\_height/div);

}

\*par\_width = \_par\_width;

\*par\_height = \_par\_height;

return;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* IPB frame encoder entry point

\*

\* Returned values :

\* - >0 - output bytes

\* - 0 - no output

\* - XVID\_ERR\_VERSION - wrong version passed to core

\* - XVID\_ERR\_END - End of stream reached before end of coding

\* - XVID\_ERR\_FORMAT - the image subsystem reported the image had a wrong

\* format

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int

enc\_encode(Encoder \* pEnc,

xvid\_enc\_frame\_t \* xFrame,

xvid\_enc\_stats\_t \* stats,

EmbeddingPara\* pEmPara)//����Ƕ�����

{

xvid\_enc\_frame\_t \* frame;

int type;

Bitstream bs;

if (XVID\_VERSION\_MAJOR(xFrame->version) != 1 || (stats && XVID\_VERSION\_MAJOR(stats->version) != 1)) /\* v1.x.x \*/

return XVID\_ERR\_VERSION;

xFrame->out\_flags = 0;

start\_global\_timer();

BitstreamInit(&bs, xFrame->bitstream, 0);

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* enqueue image to the encoding-queue

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

if (xFrame->input.csp != XVID\_CSP\_NULL)

{

QUEUEINFO \* q = &pEnc->queue[pEnc->queue\_tail];

start\_timer();

if (image\_input

(&q->image, pEnc->mbParam.width, pEnc->mbParam.height,

pEnc->mbParam.edged\_width, (uint8\_t\*\*)xFrame->input.plane, xFrame->input.stride,

xFrame->input.csp, xFrame->vol\_flags & XVID\_VOL\_INTERLACING))

{

emms();

return XVID\_ERR\_FORMAT;

}

stop\_conv\_timer();

if ((xFrame->vop\_flags & XVID\_VOP\_CHROMAOPT)) {

image\_chroma\_optimize(&q->image,

pEnc->mbParam.width, pEnc->mbParam.height, pEnc->mbParam.edged\_width);

}

q->frame = \*xFrame;

if (xFrame->quant\_intra\_matrix)

{

memcpy(q->quant\_intra\_matrix, xFrame->quant\_intra\_matrix, 64\*sizeof(unsigned char));

q->frame.quant\_intra\_matrix = q->quant\_intra\_matrix;

}

if (xFrame->quant\_inter\_matrix)

{

memcpy(q->quant\_inter\_matrix, xFrame->quant\_inter\_matrix, 64\*sizeof(unsigned char));

q->frame.quant\_inter\_matrix = q->quant\_inter\_matrix;

}

pEnc->queue\_tail = (pEnc->queue\_tail + 1) % (pEnc->mbParam.max\_bframes+1);

pEnc->queue\_size++;

}

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* bframe flush code

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

repeat:

if (pEnc->flush\_bframes)

{

if (pEnc->bframenum\_head < pEnc->bframenum\_tail) {

DPRINTF(XVID\_DEBUG\_DEBUG,"\*\*\* BFRAME (flush) bf: head=%i tail=%i queue: head=%i tail=%i size=%i\n",

pEnc->bframenum\_head, pEnc->bframenum\_tail,

pEnc->queue\_head, pEnc->queue\_tail, pEnc->queue\_size);

if ((pEnc->mbParam.plugin\_flags & XVID\_REQORIGINAL)) {

image\_copy(&pEnc->sOriginal2, &pEnc->bframes[pEnc->bframenum\_head]->image,

pEnc->mbParam.edged\_width, pEnc->mbParam.height);

}

FrameCodeB(pEnc, pEnc->bframes[pEnc->bframenum\_head], &bs);

call\_plugins(pEnc, pEnc->bframes[pEnc->bframenum\_head], &pEnc->sOriginal2, XVID\_PLG\_AFTER, NULL, NULL, stats);

pEnc->bframenum\_head++;

goto done;

}

/\* write an empty marker to the bitstream.

for divx5 decoder compatibility, this marker must consist

of a not-coded p-vop, with a time\_base of zero, and time\_increment

indentical to the future-referece frame.

\*/

if ((pEnc->mbParam.global\_flags & XVID\_GLOBAL\_PACKED && pEnc->bframenum\_tail > 0)) {

int tmp;

int bits;

DPRINTF(XVID\_DEBUG\_DEBUG,"\*\*\* EMPTY bf: head=%i tail=%i queue: head=%i tail=%i size=%i\n",

pEnc->bframenum\_head, pEnc->bframenum\_tail,

pEnc->queue\_head, pEnc->queue\_tail, pEnc->queue\_size);

bits = BitstreamPos(&bs);

tmp = pEnc->current->seconds;

pEnc->current->seconds = 0; /\* force time\_base = 0 \*/

BitstreamWriteVopHeader(&bs, &pEnc->mbParam, pEnc->current, 0, pEnc->current->quant);

BitstreamPad(&bs);

pEnc->current->seconds = tmp;

/\* add the not-coded length to the reference frame size \*/

pEnc->current->length += (BitstreamPos(&bs) - bits) / 8;

call\_plugins(pEnc, pEnc->current, &pEnc->sOriginal, XVID\_PLG\_AFTER, NULL, NULL, stats);

/\* flush complete: reset counters \*/

pEnc->flush\_bframes = 0;

pEnc->bframenum\_head = pEnc->bframenum\_tail = 0;

goto done;

}

/\* flush complete: reset counters \*/

pEnc->flush\_bframes = 0;

pEnc->bframenum\_head = pEnc->bframenum\_tail = 0;

}

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* dequeue frame from the encoding queue

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

if (pEnc->queue\_size == 0) /\* empty \*/

{

if (xFrame->input.csp == XVID\_CSP\_NULL) /\* no futher input \*/

{

DPRINTF(XVID\_DEBUG\_DEBUG,"\*\*\* FINISH bf: head=%i tail=%i queue: head=%i tail=%i size=%i\n",

pEnc->bframenum\_head, pEnc->bframenum\_tail,

pEnc->queue\_head, pEnc->queue\_tail, pEnc->queue\_size);

if (!(pEnc->mbParam.global\_flags & XVID\_GLOBAL\_PACKED) && pEnc->mbParam.max\_bframes > 0) {

call\_plugins(pEnc, pEnc->current, &pEnc->sOriginal, XVID\_PLG\_AFTER, NULL, NULL, stats);

}

/\* if the very last frame is to be b-vop, we must change it to a p-vop \*/

if (pEnc->bframenum\_tail > 0) {

SWAP(FRAMEINFO\*, pEnc->current, pEnc->reference);

pEnc->bframenum\_tail--;

SWAP(FRAMEINFO\*, pEnc->current, pEnc->bframes[pEnc->bframenum\_tail]);

/\* convert B-VOP to P-VOP \*/

pEnc->current->quant = 100\*pEnc->current->quant - pEnc->mbParam.bquant\_offset;

pEnc->current->quant += pEnc->mbParam.bquant\_ratio - 1; /\* to avoid rouding issues \*/

pEnc->current->quant /= pEnc->mbParam.bquant\_ratio;

if ((pEnc->mbParam.plugin\_flags & XVID\_REQORIGINAL)) {

image\_copy(&pEnc->sOriginal, &pEnc->current->image,

pEnc->mbParam.edged\_width, pEnc->mbParam.height);

}

DPRINTF(XVID\_DEBUG\_DEBUG,"\*\*\* PFRAME bf: head=%i tail=%i queue: head=%i tail=%i size=%i\n",

pEnc->bframenum\_head, pEnc->bframenum\_tail,

pEnc->queue\_head, pEnc->queue\_tail, pEnc->queue\_size);

pEnc->mbParam.frame\_drop\_ratio = -1; /\* it must be a coded vop \*/

FrameCodeP(pEnc, &bs);

///////////////////////////// ò�����һ֡ʱ�����е����� ////////////////////////////////

if ((pEnc->mbParam.global\_flags & XVID\_GLOBAL\_PACKED) && pEnc->bframenum\_tail==0) {

call\_plugins(pEnc, pEnc->current, &pEnc->sOriginal, XVID\_PLG\_AFTER, NULL, NULL, stats);

}else{

pEnc->flush\_bframes = 1;

goto done;

}

}

DPRINTF(XVID\_DEBUG\_DEBUG, "\*\*\* END\n");

emms();

return XVID\_ERR\_END; /\* end of stream reached \*/

}

goto done; /\* nothing to encode yet; encoder lag \*/

}

/\* the current FRAME becomes the reference \*/

SWAP(FRAMEINFO\*, pEnc->current, pEnc->reference);

/\* remove frame from encoding-queue (head), and move it into the current \*/

image\_swap(&pEnc->current->image, &pEnc->queue[pEnc->queue\_head].image);

frame = &pEnc->queue[pEnc->queue\_head].frame;

pEnc->queue\_head = (pEnc->queue\_head + 1) % (pEnc->mbParam.max\_bframes+1);

pEnc->queue\_size--;

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* init pEnc->current fields

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

pEnc->current->fincr = pEnc->mbParam.fincr>0 ? pEnc->mbParam.fincr : frame->fincr;

inc\_frame\_num(pEnc);

pEnc->current->vol\_flags = frame->vol\_flags;

pEnc->current->vop\_flags = frame->vop\_flags;

pEnc->current->motion\_flags = frame->motion;

pEnc->current->fcode = pEnc->mbParam.m\_fcode;

pEnc->current->bcode = pEnc->mbParam.m\_fcode;

if ((xFrame->vop\_flags & XVID\_VOP\_CHROMAOPT)) {

image\_chroma\_optimize(&pEnc->current->image,

pEnc->mbParam.width, pEnc->mbParam.height, pEnc->mbParam.edged\_width);

}

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* frame type & quant selection

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

type = frame->type;

pEnc->current->quant = frame->quant;

call\_plugins(pEnc, pEnc->current, NULL, XVID\_PLG\_BEFORE, &type, (int\*)&pEnc->current->quant, stats);

if (type > 0){ /\* XVID\_TYPE\_?VOP \*/

type = type2coding(type); /\* convert XVID\_TYPE\_?VOP to bitstream coding type \*/

} else{ /\* XVID\_TYPE\_AUTO \*/

if (pEnc->iFrameNum == 0 || (pEnc->mbParam.iMaxKeyInterval > 0 && pEnc->iFrameNum >= pEnc->mbParam.iMaxKeyInterval)){

pEnc->iFrameNum = 0;

type = I\_VOP;

}else{

type = MEanalysis(&pEnc->reference->image, pEnc->current,

&pEnc->mbParam, pEnc->mbParam.iMaxKeyInterval,

pEnc->iFrameNum, pEnc->bframenum\_tail, xFrame->bframe\_threshold,

(pEnc->bframes) ? pEnc->bframes[pEnc->bframenum\_head]->mbs: NULL);

}

}

if (type != I\_VOP)

pEnc->current->vol\_flags = pEnc->mbParam.vol\_flags; /\* don't allow VOL changes here \*/

/\* bframes buffer overflow check \*/

if (type == B\_VOP && pEnc->bframenum\_tail >= pEnc->mbParam.max\_bframes) {

type = P\_VOP;

}

pEnc->iFrameNum++;

if ((pEnc->current->vop\_flags & XVID\_VOP\_DEBUG)) {

image\_printf(&pEnc->current->image, pEnc->mbParam.edged\_width, pEnc->mbParam.height, 5, 5,

"%d st:%lld if:%d", pEnc->current->frame\_num, pEnc->current->stamp, pEnc->iFrameNum);

}

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* encode this frame as a b-vop

\* (we dont encode here, rather we store the frame in the bframes queue, to be encoded later)

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

if (type == B\_VOP) {

if ((pEnc->current->vop\_flags & XVID\_VOP\_DEBUG)) {

image\_printf(&pEnc->current->image, pEnc->mbParam.edged\_width, pEnc->mbParam.height, 5, 200, "BVOP");

}

if (frame->quant < 1) {

pEnc->current->quant = ((((pEnc->reference->quant + pEnc->current->quant) \*

pEnc->mbParam.bquant\_ratio) / 2) + pEnc->mbParam.bquant\_offset)/100;

} else {

pEnc->current->quant = frame->quant;

}

if (pEnc->current->quant < 1)

pEnc->current->quant = 1;

else if (pEnc->current->quant > 31)

pEnc->current->quant = 31;

DPRINTF(XVID\_DEBUG\_DEBUG,"\*\*\* BFRAME (store) bf: head=%i tail=%i queue: head=%i tail=%i size=%i quant=%i\n",

pEnc->bframenum\_head, pEnc->bframenum\_tail,

pEnc->queue\_head, pEnc->queue\_tail, pEnc->queue\_size,pEnc->current->quant);

/\* store frame into bframe buffer & swap ref back to current \*/

SWAP(FRAMEINFO\*, pEnc->current, pEnc->bframes[pEnc->bframenum\_tail]);

SWAP(FRAMEINFO\*, pEnc->current, pEnc->reference);

pEnc->bframenum\_tail++;

goto repeat;

}

DPRINTF(XVID\_DEBUG\_DEBUG,"\*\*\* XXXXXX bf: head=%i tail=%i queue: head=%i tail=%i size=%i\n",

pEnc->bframenum\_head, pEnc->bframenum\_tail,

pEnc->queue\_head, pEnc->queue\_tail, pEnc->queue\_size);

/\* for unpacked bframes, output the stats for the last encoded frame \*/

if (!(pEnc->mbParam.global\_flags & XVID\_GLOBAL\_PACKED) && pEnc->mbParam.max\_bframes > 0)

{

if (pEnc->current->stamp > 0) {

call\_plugins(pEnc, pEnc->reference, &pEnc->sOriginal, XVID\_PLG\_AFTER, NULL, NULL, stats);

}

else if (stats) {

stats->type = XVID\_TYPE\_NOTHING;

}

}

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* closed-gop

\* if the frame prior to an iframe is scheduled as a bframe, we must change it to a pframe

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

if (type == I\_VOP && (pEnc->mbParam.global\_flags & XVID\_GLOBAL\_CLOSED\_GOP) && pEnc->bframenum\_tail > 0) {

/\* place this frame back on the encoding-queue (head) \*/

/\* we will deal with it next time \*/

dec\_frame\_num(pEnc);

pEnc->iFrameNum--;

pEnc->queue\_head = (pEnc->queue\_head + (pEnc->mbParam.max\_bframes+1) - 1) % (pEnc->mbParam.max\_bframes+1);

pEnc->queue\_size++;

image\_swap(&pEnc->current->image, &pEnc->queue[pEnc->queue\_head].image);

/\* grab the last frame from the bframe-queue \*/

pEnc->bframenum\_tail--;

SWAP(FRAMEINFO\*, pEnc->current, pEnc->bframes[pEnc->bframenum\_tail]);

if ((pEnc->current->vop\_flags & XVID\_VOP\_DEBUG)) {

image\_printf(&pEnc->current->image, pEnc->mbParam.edged\_width, pEnc->mbParam.height, 5, 100, "CLOSED GOP BVOP->PVOP");

}

/\* convert B-VOP quant to P-VOP \*/

pEnc->current->quant = 100\*pEnc->current->quant - pEnc->mbParam.bquant\_offset;

pEnc->current->quant += pEnc->mbParam.bquant\_ratio - 1; /\* to avoid rouding issues \*/

pEnc->current->quant /= pEnc->mbParam.bquant\_ratio;

type = P\_VOP;

}

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* encode this frame as an i-vop

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

if (type == I\_VOP) {

DPRINTF(XVID\_DEBUG\_DEBUG,"\*\*\* IFRAME bf: head=%i tail=%i queue: head=%i tail=%i size=%i\n",

pEnc->bframenum\_head, pEnc->bframenum\_tail,

pEnc->queue\_head, pEnc->queue\_tail, pEnc->queue\_size);

if ((pEnc->current->vop\_flags & XVID\_VOP\_DEBUG)) {

image\_printf(&pEnc->current->image, pEnc->mbParam.edged\_width, pEnc->mbParam.height, 5, 200, "IVOP");

}

pEnc->iFrameNum = 1;

/\* ---- update vol flags at IVOP ----------- \*/

pEnc->mbParam.vol\_flags = pEnc->current->vol\_flags;

/\* Aspect ratio \*/

switch(frame->par) {

case XVID\_PAR\_11\_VGA:

case XVID\_PAR\_43\_PAL:

case XVID\_PAR\_43\_NTSC:

case XVID\_PAR\_169\_PAL:

case XVID\_PAR\_169\_NTSC:

case XVID\_PAR\_EXT:

pEnc->mbParam.par = frame->par;

break;

default:

pEnc->mbParam.par = XVID\_PAR\_11\_VGA;

break;

}

/\* For extended PAR only, we try to sanityse/simplify par values \*/

if (pEnc->mbParam.par == XVID\_PAR\_EXT) {

pEnc->mbParam.par\_width = frame->par\_width;

pEnc->mbParam.par\_height = frame->par\_height;

simplify\_par(&pEnc->mbParam.par\_width, &pEnc->mbParam.par\_height);

}

if ((pEnc->mbParam.vol\_flags & XVID\_VOL\_MPEGQUANT)) {

if (frame->quant\_intra\_matrix != NULL)

set\_intra\_matrix(pEnc->mbParam.mpeg\_quant\_matrices, frame->quant\_intra\_matrix);

if (frame->quant\_inter\_matrix != NULL)

set\_inter\_matrix(pEnc->mbParam.mpeg\_quant\_matrices, frame->quant\_inter\_matrix);

}

/\* prevent vol/vop misuse \*/

if (!(pEnc->current->vol\_flags & XVID\_VOL\_INTERLACING))

pEnc->current->vop\_flags &= ~(XVID\_VOP\_TOPFIELDFIRST|XVID\_VOP\_ALTERNATESCAN);

/\* ^^^------------------------ \*/

if ((pEnc->mbParam.plugin\_flags & XVID\_REQORIGINAL)) {

image\_copy(&pEnc->sOriginal, &pEnc->current->image,

pEnc->mbParam.edged\_width, pEnc->mbParam.height);

}

FrameCodeI(pEnc, &bs);

xFrame->out\_flags |= XVID\_KEYFRAME;

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* encode this frame as an p-vop

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

} else { /\* (type == P\_VOP || type == S\_VOP) \*/

DPRINTF(XVID\_DEBUG\_DEBUG,"\*\*\* PFRAME bf: head=%i tail=%i queue: head=%i tail=%i size=%i\n",

pEnc->bframenum\_head, pEnc->bframenum\_tail,

pEnc->queue\_head, pEnc->queue\_tail, pEnc->queue\_size);

if ((pEnc->current->vop\_flags & XVID\_VOP\_DEBUG)) {

image\_printf(&pEnc->current->image, pEnc->mbParam.edged\_width, pEnc->mbParam.height, 5, 200, "PVOP");

}

if ((pEnc->mbParam.plugin\_flags & XVID\_REQORIGINAL)) {

image\_copy(&pEnc->sOriginal, &pEnc->current->image,

pEnc->mbParam.edged\_width, pEnc->mbParam.height);

}

if ( FrameCodeP(pEnc, &bs,pEmPara) == 0 ) {/////////////// P ֡���붼���������////////////////

/\* N-VOP, we mustn't code b-frames yet \*/

if ((pEnc->mbParam.global\_flags & XVID\_GLOBAL\_PACKED) ||

pEnc->mbParam.max\_bframes == 0)

call\_plugins(pEnc, pEnc->current, &pEnc->sOriginal, XVID\_PLG\_AFTER, NULL, NULL, stats);

goto done;

}

}

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* on next enc\_encode call we must flush bframes

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

/\*done\_flush:\*/

pEnc->flush\_bframes = 1;

/\* packed & queued\_bframes: dont bother outputting stats here, we do so after the flush \*/

if ((pEnc->mbParam.global\_flags & XVID\_GLOBAL\_PACKED) && pEnc->bframenum\_tail > 0) {

goto repeat;

}

/\* packed or no-bframes or no-bframes-queued: output stats \*/

if ((pEnc->mbParam.global\_flags & XVID\_GLOBAL\_PACKED) || pEnc->mbParam.max\_bframes == 0 ) {

call\_plugins(pEnc, pEnc->current, &pEnc->sOriginal, XVID\_PLG\_AFTER, NULL, NULL, stats);

}

/\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\* done; return number of bytes consumed

\* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \*/

done:

stop\_global\_timer();

write\_timer();

emms();

return BitstreamLength(&bs);

}

static void SetMacroblockQuants(MBParam \* const pParam, FRAMEINFO \* frame)

{

//���ú����������ӣ���������ԽС������Խ��

unsigned int i;

MACROBLOCK \* pMB = frame->mbs;

int quant = frame->mbs[0].quant; /\* set by XVID\_PLG\_FRAME \*/

if (quant > 31)

frame->quant = quant = 31;

else if (quant < 1)

frame->quant = quant = 1;

for (i = 0; i < pParam->mb\_height \* pParam->mb\_width; i++) {

quant += pMB->dquant;

if (quant > 31)

quant = 31;

else if (quant < 1)

quant = 1;

pMB->quant = quant;

pMB++;

}

}

static \_\_inline void

CodeIntraMB(MACROBLOCK \* pMB)

{

pMB->mode = MODE\_INTRA;

/\* zero mv statistics \*/

pMB->mvs[0].x = pMB->mvs[1].x = pMB->mvs[2].x = pMB->mvs[3].x = 0;

pMB->mvs[0].y = pMB->mvs[1].y = pMB->mvs[2].y = pMB->mvs[3].y = 0;

pMB->sad8[0] = pMB->sad8[1] = pMB->sad8[2] = pMB->sad8[3] = 0;

pMB->sad16 = 0;

if (pMB->dquant != 0) {

pMB->mode = MODE\_INTRA\_Q;

}

}

static void

SliceCodeI(SMPData \*data)

{

Encoder \*pEnc = (Encoder \*) data->pEnc;

Bitstream \*bs = (Bitstream \*) data->bs;

uint16\_t x, y;

int mb\_width = pEnc->mbParam.mb\_width;

int mb\_height = pEnc->mbParam.mb\_height;

int bound = 0, num\_slices = pEnc->num\_slices;

FRAMEINFO \*const current = pEnc->current;

DECLARE\_ALIGNED\_MATRIX(dct\_codes, 6, 64, int16\_t, CACHE\_LINE);

DECLARE\_ALIGNED\_MATRIX(qcoeff, 6, 64, int16\_t, CACHE\_LINE);

if (data->start\_y > 0) { /\* write resync marker \*/

bound = data->start\_y\*mb\_width;

write\_video\_packet\_header(bs, &pEnc->mbParam, current, bound);

}

for (y = data->start\_y; y < data->stop\_y; y++) {

int new\_bound = mb\_width \* ((((y\*num\_slices) / mb\_height) \* mb\_height + (num\_slices-1)) / num\_slices);

if (new\_bound > bound) {

bound = new\_bound;

BitstreamPadAlways(bs);

write\_video\_packet\_header(bs, &pEnc->mbParam, current, bound);

}

for (x = 0; x < mb\_width; x++) {

MACROBLOCK \*pMB = &current->mbs[x + y \* mb\_width];

CodeIntraMB(pMB);

MBTransQuantIntra(&pEnc->mbParam, current, pMB, x, y,

dct\_codes, qcoeff);

start\_timer();

MBPrediction(current, x, y, mb\_width, qcoeff, bound);

stop\_prediction\_timer();

start\_timer();

MBCoding(current, pMB, qcoeff, bs, data->sStat);

stop\_coding\_timer();

}

}

emms();

BitstreamPadAlways(bs);

}

static \_\_inline void

SerializeBitstreams(Encoder \*pEnc, FRAMEINFO \*current, Bitstream \*bs, int num\_threads)

{

int k;

uint32\_t pos = BitstreamLength(bs);

for (k = 1; k < num\_threads; k++) {

uint32\_t len = BitstreamLength(pEnc->smpData[k].bs);

memcpy((void \*)((ptr\_t)bs->start + pos),

(void \*)((ptr\_t)pEnc->smpData[k].bs->start), len);

current->length += len;

pos += len;

/\* collect stats \*/

current->sStat.iTextBits += pEnc->smpData[k].sStat->iTextBits;

current->sStat.kblks += pEnc->smpData[k].sStat->kblks;

current->sStat.mblks += pEnc->smpData[k].sStat->mblks;

current->sStat.ublks += pEnc->smpData[k].sStat->ublks;

current->sStat.iMVBits += pEnc->smpData[k].sStat->iMVBits;

}

if (num\_threads > 1) {

uint32\_t pos32 = pos>>2;

bs->tail = bs->start + pos32;

bs->pos = 8\*(pos - (pos32<<2));

bs->buf = 0;

if (bs->pos > 0) {

uint32\_t pos8 = bs->pos/8;

memset((void \*)((ptr\_t)bs->tail+pos8), 0, (4-pos8));

pos = \*bs->tail;

#ifndef ARCH\_IS\_BIG\_ENDIAN

BSWAP(pos);

#endif

bs->buf = pos;

}

}

}

static int

FrameCodeI(Encoder \* pEnc,

Bitstream \* bs)

{

int bits = BitstreamPos(bs);

int bound = 0, num\_slices = pEnc->num\_slices;

int num\_threads = MAX(1, MIN(pEnc->num\_threads, num\_slices));

int slices\_per\_thread = (num\_slices\*1024 / num\_threads);

int mb\_height = pEnc->mbParam.mb\_height;

#ifdef HAVE\_PTHREAD

void \* status = NULL;

#endif

uint16\_t k;

pEnc->mbParam.m\_rounding\_type = 1;

pEnc->current->rounding\_type = pEnc->mbParam.m\_rounding\_type;

pEnc->current->coding\_type = I\_VOP;

call\_plugins(pEnc, pEnc->current, NULL, XVID\_PLG\_FRAME, NULL, NULL, NULL);

SetMacroblockQuants(&pEnc->mbParam, pEnc->current);

BitstreamWriteVolHeader(bs, &pEnc->mbParam, pEnc->current, num\_slices);

set\_timecodes(pEnc->current,pEnc->reference,pEnc->mbParam.fbase);

BitstreamPad(bs);

BitstreamWriteVopHeader(bs, &pEnc->mbParam, pEnc->current, 1, pEnc->current->mbs[0].quant);

pEnc->current->sStat.iTextBits = 0;

/\* multithreaded intra coding - dispatch threads \*/

for (k = 0; k < num\_threads; k++) {

int add = ((slices\_per\_thread + 512) >> 10);

slices\_per\_thread += ((num\_slices\*1024 / num\_threads) - add\*1024);

pEnc->smpData[k].pEnc = (void \*) pEnc;

pEnc->smpData[k].stop\_y = (((bound+add) \* mb\_height + (num\_slices-1)) / num\_slices);

pEnc->smpData[k].start\_y = ((bound \* mb\_height + (num\_slices-1)) / num\_slices);

bound += add;

if (k > 0) {

BitstreamReset(pEnc->smpData[k].bs);

pEnc->smpData[k].sStat->iTextBits = 0;

}

}

pEnc->smpData[0].bs = bs;

pEnc->smpData[0].sStat = &pEnc->current->sStat;

#ifdef HAVE\_PTHREAD

/\* create threads \*/

for (k = 1; k < num\_threads; k++) {

pthread\_create(&pEnc->smpData[k].handle, NULL,

(void\*)SliceCodeI, (void\*)&pEnc->smpData[k]);

}

#endif

SliceCodeI(&pEnc->smpData[0]);

#ifdef HAVE\_PTHREAD

/\* wait until all threads are finished \*/

for (k = 1; k < num\_threads; k++) {

pthread\_join(pEnc->smpData[k].handle, &status);

}

#endif

pEnc->current->length = BitstreamLength(bs) - (bits/8);

/\* reassemble the pieces together \*/

SerializeBitstreams(pEnc, pEnc->current, bs, num\_threads);

pEnc->current->sStat.iMVBits = 0;

pEnc->current->sStat.mblks = pEnc->current->sStat.ublks = 0;

pEnc->current->sStat.kblks = pEnc->mbParam.mb\_width \* pEnc->mbParam.mb\_height;

pEnc->fMvPrevSigma = -1;

pEnc->mbParam.m\_fcode = 2;

pEnc->current->is\_edged = 0; /\* not edged \*/

pEnc->current->is\_interpolated = -1; /\* not interpolated (fake rounding -1) \*/

return 1; /\* intra \*/

}

static \_\_inline void

updateFcode(Statistics \* sStat, Encoder \* pEnc)

{

float fSigma;

int iSearchRange;

if (sStat->iMvCount == 0)

sStat->iMvCount = 1;

fSigma = (float) sqrt((float) sStat->iMvSum / sStat->iMvCount);

iSearchRange = 16 << pEnc->mbParam.m\_fcode;

if ((3.0 \* fSigma > iSearchRange) && (pEnc->mbParam.m\_fcode <= 5) )

pEnc->mbParam.m\_fcode++;

else if ((5.0 \* fSigma < iSearchRange)

&& (4.0 \* pEnc->fMvPrevSigma < iSearchRange)

&& (pEnc->mbParam.m\_fcode >= 2) )

pEnc->mbParam.m\_fcode--;

pEnc->fMvPrevSigma = fSigma;

}

#define BFRAME\_SKIP\_THRESHHOLD 30

static void

SliceCodeP(SMPData \*data)

{

Encoder \*pEnc = (Encoder \*) data->pEnc;

Bitstream \*bs = (Bitstream \*) data->bs;

int x, y, k;

FRAMEINFO \*const current = pEnc->current;

FRAMEINFO \*const reference = pEnc->reference;

MBParam \* const pParam = &pEnc->mbParam;

int mb\_width = pParam->mb\_width;

int mb\_height = pParam->mb\_height;

DECLARE\_ALIGNED\_MATRIX(dct\_codes, 6, 64, int16\_t, CACHE\_LINE);

DECLARE\_ALIGNED\_MATRIX(qcoeff, 6, 64, int16\_t, CACHE\_LINE);

int bound = 0, num\_slices = pEnc->num\_slices;

if (data->start\_y > 0) { /\* write resync marker \*/

bound = data->start\_y\*mb\_width;

write\_video\_packet\_header(bs, pParam, current, bound);

}

for (y = data->start\_y; y < data->stop\_y; y++) {

int new\_bound = mb\_width \* ((((y\*num\_slices) / mb\_height) \* mb\_height + (num\_slices-1)) / num\_slices);

if (new\_bound > bound) {

bound = new\_bound;

BitstreamPadAlways(bs);

write\_video\_packet\_header(bs, pParam, current, bound);

}

for (x = 0; x < mb\_width; x++) {

MACROBLOCK \*pMB = &current->mbs[x + y \* pParam->mb\_width];

int skip\_possible;

if (pMB->mode == MODE\_INTRA || pMB->mode == MODE\_INTRA\_Q) {

CodeIntraMB(pMB);

MBTransQuantIntra(pParam, current, pMB, x, y,

dct\_codes, qcoeff);

start\_timer();

MBPrediction(current, x, y, pParam->mb\_width, qcoeff, bound);

stop\_prediction\_timer();

data->sStat->kblks++;

MBCoding(current, pMB, qcoeff, bs, data->sStat);

/////// �Կ���б��� ///////

stop\_coding\_timer();

continue;

}

start\_timer();

MBMotionCompensation(pMB, x, y, &reference->image,

&pEnc->vInterH, &pEnc->vInterV,

&pEnc->vInterHV, &pEnc->vGMC,

&current->image,

dct\_codes, pParam->width,

pParam->height,

pParam->edged\_width,

(current->vol\_flags & XVID\_VOL\_QUARTERPEL),

current->rounding\_type,

data->RefQ);

//////////// �Ա����Ŀ�����˶��������õ��ο��� //////////////

stop\_comp\_timer();

pMB->field\_pred = 0;

if (pMB->cbp != 0) {

pMB->cbp = MBTransQuantInter(pParam, current, pMB, x, y,

dct\_codes, qcoeff);

}

///////////// �ؽ���� //////////////

if (pMB->dquant != 0)

MBSetDquant(pMB, x, y, pParam);

if (pMB->cbp || pMB->mvs[0].x || pMB->mvs[0].y ||

pMB->mvs[1].x || pMB->mvs[1].y || pMB->mvs[2].x ||

pMB->mvs[2].y || pMB->mvs[3].x || pMB->mvs[3].y) {

data->sStat->mblks++;

} else {

data->sStat->ublks++;

}

start\_timer();

/\* Finished processing the MB, now check if to CODE or SKIP \*/

skip\_possible = (pMB->cbp == 0) && (pMB->mode == MODE\_INTER);

//skip\_possible = 0;

//�����������ǿ��ȡ��

if (current->coding\_type == S\_VOP)

skip\_possible &= (pMB->mcsel == 1);

else { /\* PVOP \*/

const VECTOR \* const mv = (pParam->vol\_flags & XVID\_VOL\_QUARTERPEL) ?

pMB->qmvs : pMB->mvs;

skip\_possible &= ((mv->x|mv->y) == 0);

}

if ((pMB->mode == MODE\_NOT\_CODED) || (skip\_possible)) {

/\* This is a candidate for SKIPping, but for P-VOPs check intermediate B-frames first \*/

int bSkip = 1;

if (current->coding\_type == P\_VOP) { /\* special rule for P-VOP's SKIP \*/

for (k = pEnc->bframenum\_head; k < pEnc->bframenum\_tail; k++) {

int iSAD;

iSAD = sad16(reference->image.y + 16\*y\*pParam->edged\_width + 16\*x,

pEnc->bframes[k]->image.y + 16\*y\*pParam->edged\_width + 16\*x,

pParam->edged\_width, BFRAME\_SKIP\_THRESHHOLD \* pMB->quant);

if (iSAD >= BFRAME\_SKIP\_THRESHHOLD \* pMB->quant || ((bound > 1) &&

((y\*mb\_width+x == bound) || (y\*mb\_width+x == bound+1)))) { /\* Some third-party decoders have problems with coloc skip MB before or after

resync marker in BVOP. We avoid any ambiguity and force no skip at slice boundary \*/

bSkip = 0; /\* could not SKIP \*/

////////////// �� �� �� �� �� �� �� Ӱ �� //////////////

if (pParam->vol\_flags & XVID\_VOL\_QUARTERPEL) {//4��֮һ����

VECTOR predMV = get\_qpmv2(current->mbs, pParam->mb\_width, bound, x, y, 0);

pMB->pmvs[0].x = - predMV.x;

pMB->pmvs[0].y = - predMV.y;

} else {//������

VECTOR predMV = get\_pmv2(current->mbs, pParam->mb\_width, bound, x, y, 0);

pMB->pmvs[0].x = - predMV.x;

pMB->pmvs[0].y = - predMV.y;

}

pMB->mode = MODE\_INTER;

pMB->cbp = 0;

break;

}

}

}

if (bSkip) {

/\* do SKIP \*/

pMB->mode = MODE\_NOT\_CODED;

MBSkip(bs);

stop\_coding\_timer();

continue; /\* next MB \*/

}

}

/\* ordinary case: normal coded INTER/INTER4V block \*/

MBCoding(current, pMB, qcoeff, bs, data->sStat);

/////// ������ ///////

stop\_coding\_timer();

}

}

BitstreamPadAlways(bs); /\* next\_start\_code() at the end of VideoObjectPlane() \*/

emms();

}

/\* FrameCodeP also handles S(GMC)-VOPs \*/

static int

FrameCodeP(Encoder \* pEnc, Bitstream \* bs,EmbeddingPara\* pEmPara)//����Ƕ�����

{

int bits = BitstreamPos(bs);

FRAMEINFO \*const current = pEnc->current;

FRAMEINFO \*const reference = pEnc->reference;

MBParam \* const pParam = &pEnc->mbParam;

int mb\_width = pParam->mb\_width;

int mb\_height = pParam->mb\_height;

int coded = 1;

int k = 0, bound = 0, num\_slices = pEnc->num\_slices;

int num\_threads = MAX(1, MIN(pEnc->num\_threads, num\_slices));

#ifdef HAVE\_PTHREAD

void \* status = NULL;

int threads\_per\_slice = (pEnc->num\_threads\*1024 / num\_threads);

#endif

int slices\_per\_thread = (num\_slices\*1024 / num\_threads);

IMAGE \*pRef = &reference->image;

if (!reference->is\_edged) {

start\_timer();

image\_setedges(pRef, pParam->edged\_width, pParam->edged\_height,

pParam->width, pParam->height, XVID\_BS\_VERSION);

stop\_edges\_timer();

reference->is\_edged = 1;

}

pParam->m\_rounding\_type = 1 - pParam->m\_rounding\_type;

current->rounding\_type = pParam->m\_rounding\_type;

current->fcode = pParam->m\_fcode;

if ((current->vop\_flags & XVID\_VOP\_HALFPEL)) {

if (reference->is\_interpolated != current->rounding\_type) {

start\_timer();

image\_interpolate(pRef->y, pEnc->vInterH.y, pEnc->vInterV.y,

pEnc->vInterHV.y, pParam->edged\_width,

pParam->edged\_height,

(pParam->vol\_flags & XVID\_VOL\_QUARTERPEL),

current->rounding\_type);

stop\_inter\_timer();

reference->is\_interpolated = current->rounding\_type;

}

}

current->sStat.iTextBits = current->sStat.iMvSum = current->sStat.iMvCount =

current->sStat.kblks = current->sStat.mblks = current->sStat.ublks =

current->sStat.iMVBits = 0;

current->coding\_type = P\_VOP;

if (current->vop\_flags & XVID\_VOP\_RD\_PSNRHVSM) {

image\_block\_variance(&current->image, pParam->edged\_width, current->mbs,

pParam->mb\_width, pParam->mb\_height);

}

call\_plugins(pEnc, pEnc->current, NULL, XVID\_PLG\_FRAME, NULL, NULL, NULL);

SetMacroblockQuants(&pEnc->mbParam, current);

start\_timer();

if (current->vol\_flags & XVID\_VOL\_GMC) /\* GMC only for S(GMC)-VOPs \*/

{ int gmcval;

current->warp = GlobalMotionEst( current->mbs, pParam, current, reference,

&pEnc->vInterH, &pEnc->vInterV, &pEnc->vInterHV, num\_slices);

if (current->motion\_flags & XVID\_ME\_GME\_REFINE) {

gmcval = GlobalMotionEstRefine(&current->warp,

current->mbs, pParam,

current, reference,

&current->image,

&reference->image,

&pEnc->vInterH,

&pEnc->vInterV,

&pEnc->vInterHV);

} else {

gmcval = globalSAD(&current->warp, pParam, current->mbs,

current,

&reference->image,

&current->image,

pEnc->vGMC.y);

}

gmcval += /\*current->quant\*/ 2 \* (int)(pParam->mb\_width\*pParam->mb\_height);

/\* 1st '3': 3 warpoints, 2nd '3': 16th pel res (2<<3) \*/

generate\_GMCparameters( 3, 3, &current->warp,

pParam->width, pParam->height,

&current->new\_gmc\_data);

if ( (gmcval<0) && ( (current->warp.duv[1].x != 0) || (current->warp.duv[1].y != 0) ||

(current->warp.duv[2].x != 0) || (current->warp.duv[2].y != 0) ) )

{

current->coding\_type = S\_VOP;

generate\_GMCimage(&current->new\_gmc\_data, &reference->image,

pParam->mb\_width, pParam->mb\_height,

pParam->edged\_width, pParam->edged\_width/2,

pParam->m\_fcode, ((pParam->vol\_flags & XVID\_VOL\_QUARTERPEL)?1:0), 0,

current->rounding\_type, current->mbs, &pEnc->vGMC);

} else {

generate\_GMCimage(&current->new\_gmc\_data, &reference->image,

pParam->mb\_width, pParam->mb\_height,

pParam->edged\_width, pParam->edged\_width/2,

pParam->m\_fcode, ((pParam->vol\_flags & XVID\_VOL\_QUARTERPEL)?1:0), 0,

current->rounding\_type, current->mbs, NULL); /\* no warping, just AMV \*/

}

}

#ifdef HAVE\_PTHREAD

if (pEnc->num\_threads > 0) {

/\* multithreaded motion estimation - dispatch threads \*/

while (k < pEnc->num\_threads) {

int i, add\_s = (slices\_per\_thread + 512) >> 10;

int add\_t = (threads\_per\_slice + 512) >> 10;

int start\_y = (bound \* mb\_height + (num\_slices-1)) / num\_slices;

int stop\_y = ((bound+add\_s) \* mb\_height + (num\_slices-1)) / num\_slices;

int rows\_per\_thread = (stop\_y - start\_y + add\_t - 1) / add\_t;

slices\_per\_thread += ((num\_slices\*1024 / num\_threads) - add\_s\*1024);

threads\_per\_slice += ((pEnc->num\_threads\*1024 / num\_threads) - add\_t\*1024);

for (i = 0; i < add\_t; i++) {

memset(pEnc->smpData[k+i].complete\_count\_self, 0, rows\_per\_thread \* sizeof(int));

pEnc->smpData[k+i].pEnc = (void \*) pEnc;

pEnc->smpData[k+i].y\_row = i;

pEnc->smpData[k+i].y\_step = add\_t;

pEnc->smpData[k+i].stop\_y = stop\_y;

pEnc->smpData[k+i].start\_y = start\_y;

/\* todo: sort out temp space once and for all \*/

pEnc->smpData[k+i].RefQ = (((k+i)&1) ? pEnc->vInterV.u : pEnc->vInterV.v) +

16\*((k+i)>>1)\*pParam->edged\_width;

}

pEnc->smpData[k].complete\_count\_above =

pEnc->smpData[k+add\_t-1].complete\_count\_self - 1;

bound += add\_s;

k += add\_t;

}

for (k = 1; k < pEnc->num\_threads; k++) {

pthread\_create(&pEnc->smpData[k].handle, NULL,

(void\*)MotionEstimateSMP, (void\*)&pEnc->smpData[k]);

}

MotionEstimateSMP(&pEnc->smpData[0]);

for (k = 1; k < pEnc->num\_threads; k++) {

pthread\_join(pEnc->smpData[k].handle, &status);

}

current->fcode = 0;

for (k = 0; k < pEnc->num\_threads; k++) {

current->sStat.iMvSum += pEnc->smpData[k].mvSum;

current->sStat.iMvCount += pEnc->smpData[k].mvCount;

if (pEnc->smpData[k].minfcode > current->fcode)

current->fcode = pEnc->smpData[k].minfcode;

}

} else

#endif

{

/\* regular ME \*/

MotionEstimation(pEnc,&pEnc->mbParam, current, reference,

&pEnc->vInterH, &pEnc->vInterV, &pEnc->vInterHV,

&pEnc->vGMC, 256\*4096, num\_slices,

pEmPara);//����Ƕ�����

////////////////////////////// �˶�ʸ���޸Ķ���������� ////////////////////////////////////

}

stop\_motion\_timer();

set\_timecodes(current,reference,pParam->fbase);

BitstreamWriteVopHeader(bs, &pEnc->mbParam, current, 1, current->mbs[0].quant);

/\* multithreaded inter coding - dispatch threads \*/

bound = 0;

slices\_per\_thread = (num\_slices\*1024 / num\_threads);

for (k = 0; k < num\_threads; k++) {

int add = ((slices\_per\_thread + 512) >> 10);

slices\_per\_thread += ((num\_slices\*1024 / num\_threads) - add\*1024);

pEnc->smpData[k].pEnc = (void \*) pEnc;

pEnc->smpData[k].stop\_y = (((bound+add) \* mb\_height + (num\_slices-1)) / num\_slices);

pEnc->smpData[k].start\_y = ((bound \* mb\_height + (num\_slices-1)) / num\_slices);

pEnc->smpData[k].RefQ = ((k&1) ? pEnc->vInterV.u : pEnc->vInterV.v) + 16\*(k>>1)\*pParam->edged\_width;

bound += add;

if (k > 0) {

pEnc->smpData[k].sStat->iTextBits = pEnc->smpData[k].sStat->kblks =

pEnc->smpData[k].sStat->mblks = pEnc->smpData[k].sStat->ublks =

pEnc->smpData[k].sStat->iMVBits = 0;

BitstreamReset(pEnc->smpData[k].bs);

}

}

pEnc->smpData[0].bs = bs;

pEnc->smpData[0].sStat = &current->sStat;

#ifdef HAVE\_PTHREAD

/\* create threads \*/

for (k = 1; k < num\_threads; k++) {

pthread\_create(&pEnc->smpData[k].handle, NULL,

(void\*)SliceCodeP, (void\*)&pEnc->smpData[k]);

}

#endif

SliceCodeP(&pEnc->smpData[0]);

#ifdef HAVE\_PTHREAD

/\* wait until all threads are finished \*/

for (k = 1; k < num\_threads; k++) {

pthread\_join(pEnc->smpData[k].handle, &status);

}

#endif

current->length = BitstreamLength(bs) - (bits/8);

/\* reassemble the pieces together \*/

SerializeBitstreams(pEnc, pEnc->current, bs, num\_threads);

updateFcode(&current->sStat, pEnc);

/\* frame drop code \*/

#if 0

DPRINTF(XVID\_DEBUG\_DEBUG, "kmu %i %i %i\n", current->sStat.kblks, current->sStat.mblks, current->sStat.ublks);

#endif

if (current->sStat.kblks + current->sStat.mblks <

(pParam->frame\_drop\_ratio \* mb\_width \* mb\_height) / 100 &&

( (pEnc->bframenum\_head >= pEnc->bframenum\_tail) || !(pEnc->mbParam.global\_flags & XVID\_GLOBAL\_CLOSED\_GOP)) &&

(current->coding\_type == P\_VOP) )

{

current->sStat.kblks = current->sStat.mblks = current->sStat.iTextBits = 0;

current->sStat.ublks = mb\_width \* mb\_height;

BitstreamReset(bs);

set\_timecodes(current,reference,pParam->fbase);

BitstreamWriteVopHeader(bs, &pEnc->mbParam, current, 0, current->mbs[0].quant);

/\* copy reference frame details into the current frame \*/

current->quant = reference->quant;

current->motion\_flags = reference->motion\_flags;

current->rounding\_type = reference->rounding\_type;

current->fcode = reference->fcode;

current->bcode = reference->bcode;

current->stamp = reference->stamp;

image\_copy(&current->image, &reference->image, pParam->edged\_width, pParam->height);

memcpy(current->mbs, reference->mbs, sizeof(MACROBLOCK) \* mb\_width \* mb\_height);

coded = 0;

BitstreamPadAlways(bs); /\* next\_start\_code() at the end of VideoObjectPlane() \*/

current->length = (BitstreamPos(bs) - bits) / 8;

} else {

pEnc->current->is\_edged = 0; /\* not edged \*/

pEnc->current->is\_interpolated = -1; /\* not interpolated (fake rounding -1) \*/

/\* what was this frame's interpolated reference will become

forward (past) reference in b-frame coding \*/

image\_swap(&pEnc->vInterH, &pEnc->f\_refh);

image\_swap(&pEnc->vInterV, &pEnc->f\_refv);

image\_swap(&pEnc->vInterHV, &pEnc->f\_refhv);

}

/\* XXX: debug

{

char s[100];

sprintf(s, "\\%05i\_cur.pgm", pEnc->m\_framenum);

image\_dump\_yuvpgm(&current->image,

pParam->edged\_width,

pParam->width, pParam->height, s);

sprintf(s, "\\%05i\_ref.pgm", pEnc->m\_framenum);

image\_dump\_yuvpgm(&reference->image,

pParam->edged\_width,

pParam->width, pParam->height, s);

}

\*/

return coded;

}

static void

SliceCodeB(SMPData \*data)

{

Encoder \*pEnc = (Encoder \*) data->pEnc;

Bitstream \*bs = (Bitstream \*) data->bs;

DECLARE\_ALIGNED\_MATRIX(dct\_codes, 6, 64, int16\_t, CACHE\_LINE);

DECLARE\_ALIGNED\_MATRIX(qcoeff, 6, 64, int16\_t, CACHE\_LINE);

int x, y;

FRAMEINFO \* const frame = (FRAMEINFO \* const) data->current;

MBParam \* const pParam = &pEnc->mbParam;

int mb\_width = pParam->mb\_width;

int mb\_height = pParam->mb\_height;

IMAGE \*f\_ref = &pEnc->reference->image;

IMAGE \*b\_ref = &pEnc->current->image;

int bound = data->start\_y\*mb\_width;

int num\_slices = pEnc->num\_slices;

if (data->start\_y > 0) { /\* write resync marker \*/

write\_video\_packet\_header(bs, pParam, frame, bound+1);

}

for (y = data->start\_y; y < MIN(data->stop\_y+1, mb\_height); y++) {

int new\_bound = mb\_width \* ((((y\*num\_slices) / mb\_height) \* mb\_height + (num\_slices-1)) / num\_slices);

int stop\_x = (y == data->stop\_y) ? 1 : mb\_width;

int start\_x = (y == data->start\_y && y > 0) ? 1 : 0;

for (x = start\_x; x < stop\_x; x++) {

MACROBLOCK \* const mb = &frame->mbs[x + y \* pEnc->mbParam.mb\_width];

/\* decoder ignores mb when refence block is INTER(0,0), CBP=0 \*/

if (mb->mode == MODE\_NOT\_CODED) {

if (pParam->plugin\_flags & XVID\_REQORIGINAL) {

MBMotionCompensation(mb, x, y, f\_ref, NULL, f\_ref, NULL, NULL, &frame->image,

NULL, 0, 0, pParam->edged\_width, 0, 0, data->RefQ);

}

continue;

}

if (new\_bound > bound && x > 0) {

bound = new\_bound;

BitstreamPadAlways(bs);

write\_video\_packet\_header(bs, pParam, frame, y\*mb\_width+x);

}

mb->quant = frame->quant;

if (mb->cbp != 0 || pParam->plugin\_flags & XVID\_REQORIGINAL) {

/\* we have to motion-compensate, transfer etc,

because there might be blocks to code \*/

MBMotionCompensationBVOP(pParam, mb, x, y, &frame->image,

f\_ref, &pEnc->f\_refh, &pEnc->f\_refv,

&pEnc->f\_refhv, b\_ref, &pEnc->vInterH,

&pEnc->vInterV, &pEnc->vInterHV, dct\_codes,

data->RefQ);

mb->cbp = MBTransQuantInterBVOP(pParam, frame, mb, x, y, dct\_codes, qcoeff);

}

if (mb->mode == MODE\_DIRECT\_NO4V)

mb->mode = MODE\_DIRECT;

if (mb->mode == MODE\_DIRECT && (mb->cbp | mb->pmvs[3].x | mb->pmvs[3].y) == 0)

mb->mode = MODE\_DIRECT\_NONE\_MV; /\* skipped \*/

else

if (frame->vop\_flags & XVID\_VOP\_GREYSCALE)

/\* keep only bits 5-2 -- Chroma blocks will just be skipped by MBCodingBVOP \*/

mb->cbp &= 0x3C;

start\_timer();

MBCodingBVOP(frame, mb, qcoeff, frame->fcode, frame->bcode, bs, data->sStat);

stop\_coding\_timer();

}

}

BitstreamPadAlways(bs); /\* next\_start\_code() at the end of VideoObjectPlane() \*/

emms();

}

static void

FrameCodeB(Encoder \* pEnc,

FRAMEINFO \* frame,

Bitstream \* bs)

{

int bits = BitstreamPos(bs);

int k = 0, bound = 0, num\_slices = pEnc->num\_slices;

int num\_threads = MAX(1, MIN(pEnc->num\_threads, num\_slices));

#ifdef HAVE\_PTHREAD

void \* status = NULL;

int threads\_per\_slice = (pEnc->num\_threads\*1024 / num\_threads);

#endif

int slices\_per\_thread = (num\_slices\*1024 / num\_threads);

IMAGE \*f\_ref = &pEnc->reference->image;

IMAGE \*b\_ref = &pEnc->current->image;

MBParam \* const pParam = &pEnc->mbParam;

int mb\_height = pParam->mb\_height;

#ifdef BFRAMES\_DEC\_DEBUG

FILE \*fp;

static char first=0;

#define BFRAME\_DEBUG if (!first && fp){ \

fprintf(fp,"Y=%3d X=%3d MB=%2d CBP=%02X\n",y,x,mb->mode,mb->cbp); \

}

if (!first){

fp=fopen("C:\\XVIDDBGE.TXT","w");

}

#endif

/\* forward \*/

if (!pEnc->reference->is\_edged) {

image\_setedges(f\_ref, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height, pEnc->mbParam.width,

pEnc->mbParam.height, XVID\_BS\_VERSION);

pEnc->reference->is\_edged = 1;

}

if (pEnc->reference->is\_interpolated != 0) {

start\_timer();

image\_interpolate(f\_ref->y, pEnc->f\_refh.y, pEnc->f\_refv.y, pEnc->f\_refhv.y,

pEnc->mbParam.edged\_width, pEnc->mbParam.edged\_height,

(pEnc->mbParam.vol\_flags & XVID\_VOL\_QUARTERPEL), 0);

stop\_inter\_timer();

pEnc->reference->is\_interpolated = 0;

}

/\* backward \*/

if (!pEnc->current->is\_edged) {

image\_setedges(b\_ref, pEnc->mbParam.edged\_width,

pEnc->mbParam.edged\_height, pEnc->mbParam.width,

pEnc->mbParam.height, XVID\_BS\_VERSION);

pEnc->current->is\_edged = 1;

}

if (pEnc->current->is\_interpolated != 0) {

start\_timer();

image\_interpolate(b\_ref->y, pEnc->vInterH.y, pEnc->vInterV.y, pEnc->vInterHV.y,

pEnc->mbParam.edged\_width, pEnc->mbParam.edged\_height,

(pEnc->mbParam.vol\_flags & XVID\_VOL\_QUARTERPEL), 0);

stop\_inter\_timer();

pEnc->current->is\_interpolated = 0;

}

frame->coding\_type = B\_VOP;

if ((frame->vop\_flags & XVID\_VOP\_RD\_PSNRHVSM) && (frame->vop\_flags & XVID\_VOP\_RD\_BVOP)) {

image\_block\_variance(&frame->image, pEnc->mbParam.edged\_width, frame->mbs,

pEnc->mbParam.mb\_width, pEnc->mbParam.mb\_height);

}

call\_plugins(pEnc, frame, NULL, XVID\_PLG\_FRAME, NULL, NULL, NULL);

frame->fcode = frame->bcode = pEnc->current->fcode;

start\_timer();

#ifdef HAVE\_PTHREAD

if (pEnc->num\_threads > 0) {

/\* multithreaded motion estimation - dispatch threads \*/

while (k < pEnc->num\_threads) {

int i, add\_s = (slices\_per\_thread + 512) >> 10;

int add\_t = (threads\_per\_slice + 512) >> 10;

int start\_y = (bound \* mb\_height + (num\_slices-1)) / num\_slices;

int stop\_y = ((bound+add\_s) \* mb\_height + (num\_slices-1)) / num\_slices;

int rows\_per\_thread = (stop\_y - start\_y + add\_t - 1) / add\_t;

slices\_per\_thread += ((num\_slices\*1024 / num\_threads) - add\_s\*1024);

threads\_per\_slice += ((pEnc->num\_threads\*1024 / num\_threads) - add\_t\*1024);

for (i = 0; i < add\_t; i++) {

memset(pEnc->smpData[k+i].complete\_count\_self, 0, rows\_per\_thread \* sizeof(int));

pEnc->smpData[k+i].pEnc = (void \*) pEnc;

pEnc->smpData[k+i].current = frame;

pEnc->smpData[k+i].y\_row = i;

pEnc->smpData[k+i].y\_step = add\_t;

pEnc->smpData[k+i].stop\_y = stop\_y;

pEnc->smpData[k+i].start\_y = start\_y;

/\* todo: sort out temp space once and for all \*/

pEnc->smpData[k+i].RefQ = (((k+i)&1) ? pEnc->vInterV.u : pEnc->vInterV.v) +

16\*((k+i)>>1)\*pParam->edged\_width;

}

pEnc->smpData[k].complete\_count\_above =

pEnc->smpData[k+add\_t-1].complete\_count\_self - 1;

bound += add\_s;

k += add\_t;

}

for (k = 1; k < pEnc->num\_threads; k++) {

pthread\_create(&pEnc->smpData[k].handle, NULL,

(void\*)SMPMotionEstimationBVOP, (void\*)&pEnc->smpData[k]);

}

SMPMotionEstimationBVOP(&pEnc->smpData[0]);

for (k = 1; k < pEnc->num\_threads; k++) {

pthread\_join(pEnc->smpData[k].handle, &status);

}

frame->fcode = frame->bcode = 0;

for (k = 0; k < pEnc->num\_threads; k++) {

if (pEnc->smpData[k].minfcode > frame->fcode)

frame->fcode = pEnc->smpData[k].minfcode;

if (pEnc->smpData[k].minbcode > frame->bcode)

frame->bcode = pEnc->smpData[k].minbcode;

}

} else

#endif

{

MotionEstimationBVOP(&pEnc->mbParam, frame,

((int32\_t)(pEnc->current->stamp - frame->stamp)), /\* time\_bp \*/

((int32\_t)(pEnc->current->stamp - pEnc->reference->stamp)), /\* time\_pp \*/

pEnc->reference->mbs, f\_ref,

&pEnc->f\_refh, &pEnc->f\_refv, &pEnc->f\_refhv,

pEnc->current, b\_ref, &pEnc->vInterH,

&pEnc->vInterV, &pEnc->vInterHV,

pEnc->num\_slices);

}

stop\_motion\_timer();

set\_timecodes(frame, pEnc->reference,pEnc->mbParam.fbase);

BitstreamWriteVopHeader(bs, &pEnc->mbParam, frame, 1, frame->quant);

/\* reset stats \*/

frame->sStat.iTextBits = 0;

frame->sStat.iMVBits = 0;

frame->sStat.iMvSum = 0;

frame->sStat.iMvCount = 0;

frame->sStat.kblks = frame->sStat.mblks = frame->sStat.ublks = 0;

frame->sStat.mblks = pEnc->mbParam.mb\_width \* pEnc->mbParam.mb\_height;

frame->sStat.kblks = frame->sStat.ublks = 0;

/\* multithreaded inter coding - dispatch threads \*/

bound = 0;

slices\_per\_thread = (num\_slices\*1024 / num\_threads);

for (k = 0; k < num\_threads; k++) {

int add = ((slices\_per\_thread + 512) >> 10);

slices\_per\_thread += ((num\_slices\*1024 / num\_threads) - add\*1024);

pEnc->smpData[k].pEnc = (void \*) pEnc;

pEnc->smpData[k].current = frame;

pEnc->smpData[k].stop\_y = (((bound+add) \* mb\_height + (num\_slices-1)) / num\_slices);

pEnc->smpData[k].start\_y = ((bound \* mb\_height + (num\_slices-1)) / num\_slices);

bound += add;

/\* todo: sort out temp space once and for all \*/

pEnc->smpData[k].RefQ = ((k&1) ? pEnc->vInterV.u : pEnc->vInterV.v) + 16\*(k>>1)\*pParam->edged\_width;

if (k > 0) {

BitstreamReset(pEnc->smpData[k].bs);

pEnc->smpData[k].sStat->iTextBits = pEnc->smpData[k].sStat->kblks =

pEnc->smpData[k].sStat->mblks = pEnc->smpData[k].sStat->ublks = pEnc->smpData[k].sStat->iMVBits = 0;

}

}

#ifdef HAVE\_PTHREAD

for (k = 1; k < num\_threads; k++) {

pthread\_create(&pEnc->smpData[k].handle, NULL,

(void\*)SliceCodeB, (void\*)&pEnc->smpData[k]);

}

#endif

pEnc->smpData[0].bs = bs;

pEnc->smpData[0].sStat = &frame->sStat;

SliceCodeB(&pEnc->smpData[0]);

#ifdef HAVE\_PTHREAD

for (k = 1; k < num\_threads; k++) {

pthread\_join(pEnc->smpData[k].handle, &status);

}

#endif

frame->length = BitstreamLength(bs) - (bits/8);

/\* reassemble the pieces together \*/

SerializeBitstreams(pEnc, frame, bs, num\_threads);

#ifdef BFRAMES\_DEC\_DEBUG

if (!first){

first=1;

if (fp)

fclose(fp);

}

#endif

}