```
(C) 2018 WINDGO Inc.
FILE: Î¼JPEG - C Source
NAME: ujpeg.c
DATE: 2018/01/12
TIME: 11:00:13
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// uJPEG (MicroJPEG) -- KeyJ's Small Baseline JPEG Decoder
// based on NanoJPEG -- KeyJ's Tiny Baseline JPEG Decoder
// version 1.3 (2012-03-05)
// by Martin J. Fiedler <martin.fiedler@gmx.net>
//
// This software is published under the terms of KeyJ's Research License,
// version 0.2. Usage of this software is subject to the following conditions:
// 0. There's no warranty whatsoever. The author(s) of this software can not
//    be held liable for any damages that occur when using this software.
// 1. This software may be used freely for both non-commercial and commercial
//    purposes.
// 2. This software may be redistributed freely as long as no fees are charged
//    for the distribution and this license information is included.
// 3. This software may be modified freely except for this license information,
//    which must not be changed in any way.
// 4. If anything other than configuration, indentation or comments have been
//    altered in the code, the original author(s) must receive a copy of the
//    modified code.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ujpeg.h"

/* UJ_NODECODE_BLOCK_SIZE: if #defined, this specifies the amount of bytes
 * to load from disk if ujDecodeFile() is used after ujDisableDecoding().
 * This will speed up checking of large files, because not the whole file has
 * to be read, but just a small portion of it. On the other hand, it will also
 * break checking of files where the actual image data starts after this point
 * (though it's questionable if any real-world file would ever trigger that). */
#define UJ_NODECODE_BLOCK_SIZE (256 * 1024)

#ifdef _MSC_VER
    #define UJ_INLINE static __inline
    #define UJ_FORCE_INLINE static __forceinline
#else
    #define UJ_INLINE static inline
    #define UJ_FORCE_INLINE static inline
#endif

typedef struct _uj_code {
    unsigned char bits, code;
} ujVLCCode;

typedef struct _uj_cmp {
    int width, height;
    int stride;
    unsigned char *pixels;
    int cid;
    int ssx, ssy;
    int qtsel;
    int actabsel, dctabsel;
    int dcpred;
} ujComponent;
```

```
typedef struct _uj_ctx {
    const unsigned char *pos;
    int valid, decoded;
    int no_decode;
    int fast_chroma;
    int size;
    int length;
    int width, height;
    int mbwidth, mbheight;
    int mbsizex, mbsizey;
    int ncomp;
    ujComponent comp[3];
    int qtused, qtavail;
    unsigned char qtab[4][64];
    ujVLCCode vlctab[4][65536];
    int buf, bufbits;
    int block[64];
    int rstinterval;
    unsigned char *rgb;
    int exif_le;
    int co_sited_chroma;
} ujContext;

static ujResult ujError = UJ_OK;

static const char ujZZ[64] = { 0, 1, 8, 16, 9, 2, 3, 10, 17, 24, 32, 25, 18,
11, 4, 5, 12, 19, 26, 33, 40, 48, 41, 34, 27, 20, 13, 6, 7, 14, 21, 28, 35,
42, 49, 56, 57, 50, 43, 36, 29, 22, 15, 23, 30, 37, 44, 51, 58, 59, 52, 45,
38, 31, 39, 46, 53, 60, 61, 54, 47, 55, 62, 63 };

UJ_FORCE_INLINE unsigned char ujClip(const int x) {
    return (x < 0) ? 0 : ((x > 0xFF) ? 0xFF : (unsigned char) x);
}

////////////////////////////////////////////////////////////////////////////////

#define W1 2841
#define W2 2676
#define W3 2408
#define W5 1609
#define W6 1108
#define W7 565

UJ_INLINE void ujRowIDCT(int* blk) {
    int x0, x1, x2, x3, x4, x5, x6, x7, x8;
    if (!((x1 = blk[4] << 11)
        | (x2 = blk[6])
        | (x3 = blk[2])
        | (x4 = blk[1])
        | (x5 = blk[7])
        | (x6 = blk[5])
        | (x7 = blk[3])))
    {
        blk[0] = blk[1] = blk[2] = blk[3] = blk[4] = blk[5] = blk[6] = blk[7] = blk[0]
<< 3;
        return;
    }
    x0 = (blk[0] << 11) + 128;
    x8 = W7 * (x4 + x5);
    x4 = x8 + (W1 - W7) * x4;
    x5 = x8 - (W1 + W7) * x5;
    x8 = W3 * (x6 + x7);
    x6 = x8 - (W3 - W5) * x6;
```

```c
    x7 = x8 - (W3 + W5) * x7;
    x8 = x0 + x1;
    x0 -= x1;
    x1 = W6 * (x3 + x2);
    x2 = x1 - (W2 + W6) * x2;
    x3 = x1 + (W2 - W6) * x3;
    x1 = x4 + x6;
    x4 -= x6;
    x6 = x5 + x7;
    x5 -= x7;
    x7 = x8 + x3;
    x8 -= x3;
    x3 = x0 + x2;
    x0 -= x2;
    x2 = (181 * (x4 + x5) + 128) >> 8;
    x4 = (181 * (x4 - x5) + 128) >> 8;
    blk[0] = (x7 + x1) >> 8;
    blk[1] = (x3 + x2) >> 8;
    blk[2] = (x0 + x4) >> 8;
    blk[3] = (x8 + x6) >> 8;
    blk[4] = (x8 - x6) >> 8;
    blk[5] = (x0 - x4) >> 8;
    blk[6] = (x3 - x2) >> 8;
    blk[7] = (x7 - x1) >> 8;
}

UJ_INLINE void ujColIDCT(const int* blk, unsigned char *out, int stride) {
    int x0, x1, x2, x3, x4, x5, x6, x7, x8;
    if (!((x1 = blk[8*4] << 8)
        | (x2 = blk[8*6])
        | (x3 = blk[8*2])
        | (x4 = blk[8*1])
        | (x5 = blk[8*7])
        | (x6 = blk[8*5])
        | (x7 = blk[8*3])))
    {
        x1 = ujClip(((blk[0] + 32) >> 6) + 128);
        for (x0 = 8;  x0;  --x0) {
            *out = (unsigned char) x1;
            out += stride;
        }
        return;
    }
    x0 = (blk[0] << 8) + 8192;
    x8 = W7 * (x4 + x5) + 4;
    x4 = (x8 + (W1 - W7) * x4) >> 3;
    x5 = (x8 - (W1 + W7) * x5) >> 3;
    x8 = W3 * (x6 + x7) + 4;
    x6 = (x8 - (W3 - W5) * x6) >> 3;
    x7 = (x8 - (W3 + W5) * x7) >> 3;
    x8 = x0 + x1;
    x0 -= x1;
    x1 = W6 * (x3 + x2) + 4;
    x2 = (x1 - (W2 + W6) * x2) >> 3;
    x3 = (x1 + (W2 - W6) * x3) >> 3;
    x1 = x4 + x6;
    x4 -= x6;
    x6 = x5 + x7;
    x5 -= x7;
    x7 = x8 + x3;
    x8 -= x3;
    x3 = x0 + x2;
    x0 -= x2;
    x2 = (181 * (x4 + x5) + 128) >> 8;
    x4 = (181 * (x4 - x5) + 128) >> 8;
    *out = ujClip(((x7 + x1) >> 14) + 128);  out += stride;
    *out = ujClip(((x3 + x2) >> 14) + 128);  out += stride;
    *out = ujClip(((x0 + x4) >> 14) + 128);  out += stride;
    *out = ujClip(((x8 + x6) >> 14) + 128);  out += stride;
    *out = ujClip(((x8 - x6) >> 14) + 128);  out += stride;
    *out = ujClip(((x0 - x4) >> 14) + 128);  out += stride;
    *out = ujClip(((x3 - x2) >> 14) + 128);  out += stride;
    *out = ujClip(((x7 - x1) >> 14) + 128);
}

///////////////////////////////////////////////////////////////////////////////

#define ujThrow(e) do { ujError = e; return; } while (0)
#define ujCheckError() do { if (ujError) return; } while (0)

static int ujShowBits(ujContext *uj, int bits) {
    unsigned char newbyte;
    if (!bits) return 0;
    while (uj->bufbits < bits) {
        if (uj->size <= 0) {
            uj->buf = (uj->buf << 8) | 0xFF;
            uj->bufbits += 8;
            continue;
        }
        newbyte = *uj->pos++;
        uj->size--;
        uj->bufbits += 8;
        uj->buf = (uj->buf << 8) | newbyte;
        if (newbyte == 0xFF) {
            if (uj->size) {
                unsigned char marker = *uj->pos++;
                uj->size--;
                switch (marker) {
                    case 0x00:
                    case 0xFF:
                        break;
                    case 0xD9: uj->size = 0; break;
                    default:
                        if ((marker & 0xF8) != 0xD0)
                            ujError = UJ_SYNTAX_ERROR;
                        else {
                            uj->buf = (uj->buf << 8) | marker;
                            uj->bufbits += 8;
                        }
                }
            } else
                ujError = UJ_SYNTAX_ERROR;
        }
    }
    return (uj->buf >> (uj->bufbits - bits)) & ((1 << bits) - 1);
}

UJ_INLINE void ujSkipBits(ujContext *uj, int bits) {
    if (uj->bufbits < bits)
        (void) ujShowBits(uj, bits);
    uj->bufbits -= bits;
}

UJ_INLINE int ujGetBits(ujContext *uj, int bits) {
    int res = ujShowBits(uj, bits);
    ujSkipBits(uj, bits);
    return res;
}

UJ_INLINE void ujByteAlign(ujContext *uj) {
    uj->bufbits &= 0xF8;
}

static void ujSkip(ujContext *uj, int count) {
    uj->pos += count;
    uj->size -= count;
    uj->length -= count;
    if (uj->size < 0) ujError = UJ_SYNTAX_ERROR;
}

UJ_INLINE unsigned short ujDecode16(const unsigned char *pos) {
    return (pos[0] << 8) | pos[1];
}

static void ujDecodeLength(ujContext *uj) {
    if (uj->size < 2) ujThrow(UJ_SYNTAX_ERROR);
    uj->length = ujDecode16(uj->pos);
    if (uj->length > uj->size) ujThrow(UJ_SYNTAX_ERROR);
    ujSkip(uj, 2);
}

UJ_INLINE void ujSkipMarker(ujContext *uj) {
    ujDecodeLength(uj);
    ujSkip(uj, uj->length);
}

UJ_INLINE void ujDecodeSOF(ujContext *uj) {
    int i, ssxmax = 0, ssymax = 0, size;
    ujComponent* c;
    ujDecodeLength(uj);
    if (uj->length < 9) ujThrow(UJ_SYNTAX_ERROR);
    if (uj->pos[0] != 8) ujThrow(UJ_UNSUPPORTED);
    uj->height = ujDecode16(uj->pos+1);
    uj->width = ujDecode16(uj->pos+3);
    uj->ncomp = uj->pos[5];
    ujSkip(uj, 6);
    switch (uj->ncomp) {
        case 1:
        case 3:
            break;
        default:
            ujThrow(UJ_UNSUPPORTED);
    }
    if (uj->length < (uj->ncomp * 3)) ujThrow(UJ_SYNTAX_ERROR);
    for (i = 0, c = uj->comp;  i < uj->ncomp;  ++i, ++c) {
        c->cid = uj->pos[0];
        if (!(c->ssx = uj->pos[1] >> 4)) ujThrow(UJ_SYNTAX_ERROR);
        if (c->ssx & (c->ssx - 1)) ujThrow(UJ_UNSUPPORTED);  // non-power of two
        if (!(c->ssy = uj->pos[1] & 15)) ujThrow(UJ_SYNTAX_ERROR);
        if (c->ssy & (c->ssy - 1)) ujThrow(UJ_UNSUPPORTED);  // non-power of two
        if ((c->qtsel = uj->pos[2]) & 0xFC) ujThrow(UJ_SYNTAX_ERROR);
        ujSkip(uj, 3);
        uj->qtused |= 1 << c->qtsel;
        if (c->ssx > ssxmax) ssxmax = c->ssx;
        if (c->ssy > ssymax) ssymax = c->ssy;
    }
    if (uj->ncomp == 1) {
        c = uj->comp;
        c->ssx = c->ssy = ssxmax = ssymax = 1;
    }
    uj->mbsizex = ssxmax << 3;
    uj->mbsizey = ssymax << 3;
    uj->mbwidth = (uj->width + uj->mbsizex - 1) / uj->mbsizex;
    uj->mbheight = (uj->height + uj->mbsizey - 1) / uj->mbsizey;
    for (i = 0, c = uj->comp;  i < uj->ncomp;  ++i, ++c) {
        c->width = (uj->width * c->ssx + ssxmax - 1) / ssxmax;
        c->stride = (c->width + 7) & 0x7FFFFFF8;
        c->height = (uj->height * c->ssy + ssymax - 1) / ssymax;
        c->stride = uj->mbwidth * uj->mbsizex * c->ssx / ssxmax;
        if (((c->width < 3) && (c->ssx != ssxmax)) || ((c->height < 3) && (c->ssy != ss
ymax))) ujThrow(UJ_UNSUPPORTED);
        size = c->stride * (uj->mbheight * uj->mbsizey * c->ssy / ssymax);
        if (!uj->no_decode) {
            if (!(c->pixels = malloc(size))) ujThrow(UJ_OUT_OF_MEM);
            memset(c->pixels, 0x80, size);
        }
    }
    ujSkip(uj, uj->length);
}

UJ_INLINE void ujDecodeDHT(ujContext *uj) {
    int codelen, currcnt, remain, spread, i, j;
    ujVLCCode *vlc;
    static unsigned char counts[16];
    ujDecodeLength(uj);
    while (uj->length >= 17) {
        i = uj->pos[0];
        if (i & 0xEC) ujThrow(UJ_SYNTAX_ERROR);
        if (i & 0x02) ujThrow(UJ_UNSUPPORTED);
        i = (i | (i >> 3)) & 3;  // combined DC/AC + tableid value
        for (codelen = 1;  codelen <= 16;  ++codelen)
            counts[codelen - 1] = uj->pos[codelen];
        ujSkip(uj, 17);
        vlc = &uj->vlctab[i][0];
        remain = spread = 65536;
        for (codelen = 1;  codelen <= 16;  ++codelen) {
            spread >>= 1;
            currcnt = counts[codelen - 1];
            if (!currcnt) continue;
            if (uj->length < currcnt) ujThrow(UJ_SYNTAX_ERROR);
            remain -= currcnt << (16 - codelen);
            if (remain < 0) ujThrow(UJ_SYNTAX_ERROR);
            for (i = 0;  i < currcnt;  ++i) {
                register unsigned char code = uj->pos[i];
                for (j = spread;  j;  --j) {
                    vlc->bits = (unsigned char) codelen;
                    vlc->code = code;
                    ++vlc;
                }
            }
            ujSkip(uj, currcnt);
        }
        while (remain--) {
            vlc->bits = 0;
            ++vlc;
        }
    }
    if (uj->length) ujThrow(UJ_SYNTAX_ERROR);
}

UJ_INLINE void ujDecodeDQT(ujContext *uj) {
    int i;
    unsigned char *t;
    ujDecodeLength(uj);
    while (uj->length >= 65) {
        i = uj->pos[0];
        if (i & 0xFC) ujThrow(UJ_SYNTAX_ERROR);
        uj->qtavail |= 1 << i;
        t = &uj->qtab[i][0];
        for (i = 0;  i < 64;  ++i)
            t[i] = uj->pos[i + 1];
        ujSkip(uj, 65);
    }
    if (uj->length) ujThrow(UJ_SYNTAX_ERROR);
}

UJ_INLINE void ujDecodeDRI(ujContext *uj) {
    ujDecodeLength(uj);
    if (uj->length < 2) ujThrow(UJ_SYNTAX_ERROR);
    uj->rstinterval = ujDecode16(uj->pos);
    ujSkip(uj, uj->length);
}

static int ujGetVLC(ujContext *uj, ujVLCCode* vlc, unsigned char* code) {
    int value = ujShowBits(uj, 16);
    int bits = vlc[value].bits;
    if (!bits) { ujError = UJ_SYNTAX_ERROR; return 0; }
    ujSkipBits(uj, bits);
    value = vlc[value].code;
    if (code) *code = (unsigned char) value;
    bits = value & 15;
    if (!bits) return 0;
    value = ujGetBits(uj, bits);
    if (value < (1 << (bits - 1)))
        value += ((-1) << bits) + 1;
    return value;
}

UJ_INLINE void ujDecodeBlock(ujContext *uj, ujComponent* c, unsigned char* out) {
    unsigned char code = 0;
    int value, coef = 0;
    memset(uj->block, 0, sizeof(uj->block));
    c->dcpred += ujGetVLC(uj, &uj->vlctab[c->dctabsel][0], NULL);
    uj->block[0] = (c->dcpred) * uj->qtab[c->qtsel][0];
    do {
        value = ujGetVLC(uj, &uj->vlctab[c->actabsel][0], &code);
        if (!code) break;  // EOB
        if (!(code & 0x0F) && (code != 0xF0)) ujThrow(UJ_SYNTAX_ERROR);
        coef += (code >> 4) + 1;
        if (coef > 63) ujThrow(UJ_SYNTAX_ERROR);
        uj->block[(int) ujZZ[coef]] = value * uj->qtab[c->qtsel][coef];
    } while (coef < 63);
    for (coef = 0;  coef < 64;  coef += 8)
        ujRowIDCT(&uj->block[coef]);
    for (coef = 0;  coef < 8;  ++coef)
        ujColIDCT(&uj->block[coef], &out[coef], c->stride);
}

UJ_INLINE void ujDecodeScan(ujContext *uj) {
    int i, mbx, mby, sbx, sby;
    int rstcount = uj->rstinterval, nextrst = 0;
    ujComponent* c;
    ujDecodeLength(uj);
    if (uj->length < (4 + 2 * uj->ncomp)) ujThrow(UJ_SYNTAX_ERROR);
    if (uj->pos[0] != uj->ncomp) ujThrow(UJ_UNSUPPORTED);
```

```c
    ujSkip(uj, 1);
    for (i = 0, c = uj->comp;  i < uj->ncomp;  ++i, ++c) {
        if (uj->pos[0] != c->cid) ujThrow(UJ_SYNTAX_ERROR);
        if (uj->pos[1] & 0xEE) ujThrow(UJ_SYNTAX_ERROR);
        c->dctabsel = uj->pos[1] >> 4;
        c->actabsel = (uj->pos[1] & 1) | 2;
        ujSkip(uj, 2);
    }
    if (uj->pos[0] || (uj->pos[1] != 63) || uj->pos[2]) ujThrow(UJ_UNSUPPORTED);
    ujSkip(uj, uj->length);
    uj->valid = 1;
    if (uj->no_decode) { ujError = __UJ_FINISHED; return; }
    uj->decoded = 1;  // mark the image as decoded now -- every subsequent error
                      // just means that the image hasn't been decoded
                      // completely
    for (mbx = mby = 0;;) {
        for (i = 0, c = uj->comp;  i < uj->ncomp;  ++i, ++c)
            for (sby = 0;  sby < c->ssy;  ++sby)
                for (sbx = 0;  sbx < c->ssx;  ++sbx) {
                    ujDecodeBlock(uj, c, &c->pixels[((mby * c->ssy + sby) * c->stride +
 mbx * c->ssx + sbx) << 3]);
                    ujCheckError();
                }
        if (++mbx >= uj->mbwidth) {
            mbx = 0;
            if (++mby >= uj->mbheight) break;
        }
        if (uj->rstinterval && !(--rstcount)) {
            ujByteAlign(uj);
            i = ujGetBits(uj, 16);
            if (((i & 0xFFF8) != 0xFFD0) || ((i & 7) != nextrst))
                ujThrow(UJ_SYNTAX_ERROR);
            nextrst = (nextrst + 1) & 7;
            rstcount = uj->rstinterval;
            for (i = 0;  i < 3;  ++i)
                uj->comp[i].dcpred = 0;
        }
    }
    ujError = __UJ_FINISHED;
}

////////////////////////////////////////////////////////////////////////////////

#define CF4A (-9)
#define CF4B (111)
#define CF4C (29)
#define CF4D (-3)
#define CF3A (28)
#define CF3B (109)
#define CF3C (-9)
#define CF3X (104)
#define CF3Y (27)
#define CF3Z (-3)
#define CF2A (139)
#define CF2B (-11)
#define CF(x) ujClip(((x) + 64) >> 7)

UJ_INLINE void ujUpsampleHCentered(ujComponent* c) {
    const int xmax = c->width - 3;
    unsigned char *out, *lin, *lout;
    int x, y;
    out = malloc((c->width * c->height) << 1);
    if (!out) ujThrow(UJ_OUT_OF_MEM);
    lin = c->pixels;
    lout = out;
    for (y = c->height;  y;  --y) {
        lout[0] = CF(CF2A * lin[0] + CF2B * lin[1]);
        lout[1] = CF(CF3X * lin[0] + CF3Y * lin[1] + CF3Z * lin[2]);
        lout[2] = CF(CF3A * lin[0] + CF3B * lin[1] + CF3C * lin[2]);
        for (x = 0;  x < xmax;  ++x) {
            lout[(x << 1) + 3] = CF(CF4A * lin[x] + CF4B * lin[x + 1] + CF4C * lin[x +
2] + CF4D * lin[x + 3]);
            lout[(x << 1) + 4] = CF(CF4D * lin[x] + CF4C * lin[x + 1] + CF4B * lin[x +
2] + CF4A * lin[x + 3]);
        }
        lin += c->stride;
        lout += c->width << 1;
        lout[-3] = CF(CF3A * lin[-1] + CF3B * lin[-2] + CF3C * lin[-3]);
        lout[-2] = CF(CF3X * lin[-1] + CF3Y * lin[-2] + CF3Z * lin[-3]);
        lout[-1] = CF(CF2A * lin[-1] + CF2B * lin[-2]);
    }
    c->width <<= 1;
    c->stride = c->width;
    free(c->pixels);
    c->pixels = out;
}

UJ_INLINE void ujUpsampleVCentered(ujComponent* c) {
    const int w = c->width, s1 = c->stride, s2 = s1 + s1;
    unsigned char *out, *cin, *cout;
    int x, y;
    out = malloc((c->width * c->height) << 1);
    if (!out) ujThrow(UJ_OUT_OF_MEM);
    for (x = 0;  x < w;  ++x) {
        cin = &c->pixels[x];
        cout = &out[x];
        *cout = CF(CF2A * cin[0] + CF2B * cin[s1]);  cout += w;
        *cout = CF(CF3X * cin[0] + CF3Y * cin[s1] + CF3Z * cin[s2]);  cout += w;
        *cout = CF(CF3A * cin[0] + CF3B * cin[s1] + CF3C * cin[s2]);  cout += w;
        cin += s1;
        for (y = c->height - 3;  y;  --y) {
            *cout = CF(CF4A * cin[-s1] + CF4B * cin[0] + CF4C * cin[s1] + CF4D * cin[s2
]);  cout += w;
            *cout = CF(CF4D * cin[-s1] + CF4C * cin[0] + CF4B * cin[s1] + CF4A * cin[s2
]);  cout += w;
            cin += s1;
        }
        cin += s1;
        *cout = CF(CF3A * cin[0] + CF3B * cin[-s1] + CF3C * cin[-s2]);  cout += w;
        *cout = CF(CF3X * cin[0] + CF3Y * cin[-s1] + CF3Z * cin[-s2]);  cout += w;
        *cout = CF(CF2A * cin[0] + CF2B * cin[-s1]);
    }
    c->height <<= 1;
    c->stride = c->width;
    free(c->pixels);
    c->pixels = out;
}

#define SF(x) ujClip(((x) + 8) >> 4)

UJ_INLINE void ujUpsampleHCoSited(ujComponent* c) {
    const int xmax = c->width - 1;
    unsigned char *out, *lin, *lout;
    int x, y;
    out = malloc((c->width * c->height) << 1);
    if (!out) ujThrow(UJ_OUT_OF_MEM);
    lin = c->pixels;
    lout = out;
    for (y = c->height;  y;  --y) {
        lout[0] = lin[0];
        lout[1] = SF((lin[0] << 3) + 9 * lin[1] - lin[2]);
        lout[2] = lin[1];
        for (x = 2;  x < xmax;  ++x) {
            lout[(x << 1) - 1] = SF(9 * (lin[x-1] + lin[x]) - (lin[x-2] + lin[x+1]));
            lout[x << 1] = lin[x];
        }
        lin += c->stride;
        lout += c->width << 1;
        lout[-3] = SF((lin[-1] << 3) + 9 * lin[-2] - lin[-3]);
        lout[-2] = lin[-1];
        lout[-1] = SF(17 * lin[-1] - lin[-2]);
    }
    c->width <<= 1;
    c->stride = c->width;
    free(c->pixels);
    c->pixels = out;
}

UJ_INLINE void ujUpsampleVCoSited(ujComponent* c) {
    const int w = c->width, s1 = c->stride, s2 = s1 + s1;
    unsigned char *out, *cin, *cout;
    int x, y;
    out = malloc((c->width * c->height) << 1);
    if (!out) ujThrow(UJ_OUT_OF_MEM);
    for (x = 0;  x < w;  ++x) {
        cin = &c->pixels[x];
        cout = &out[x];
        *cout = cin[0];  cout += w;
        *cout = SF((cin[0] << 3) + 9 * cin[s1] - cin[s2]);  cout += w;
        *cout = cin[s1];  cout += w;
        cin += s1;
        for (y = c->height - 3;  y;  --y) {
            *cout = SF(9 * (cin[0] + cin[s1]) - (cin[-s1] + cin[s2]));  cout += w;
            *cout = cin[s1];  cout += w;
            cin += s1;
        }
        *cout = SF((cin[s1] << 3) + 9 * cin[0] - cin[-s1]);  cout += w;
        *cout = cin[-s1];  cout += w;
        *cout = SF(17 * cin[s1] - cin[0]);
    }
    c->height <<= 1;
    c->stride = c->width;
    free(c->pixels);
    c->pixels = out;
}

UJ_INLINE void ujUpsampleFast(ujContext *uj, ujComponent* c) {
    int x, y, xshift = 0, yshift = 0;
    unsigned char *out, *lin, *lout;
    while (c->width < uj->width) { c->width <<= 1; ++xshift; }
    while (c->height < uj->height) { c->height <<= 1; ++yshift; }
    if (!xshift && !yshift) return;
    out = malloc(c->width * c->height);
    if (!out) ujThrow(UJ_OUT_OF_MEM);
    lin = c->pixels;
    lout = out;
    for (y = 0;  y < c->height;  ++y) {
        lin = &c->pixels[(y >> yshift) * c->stride];
        for (x = 0;  x < c->width;  ++x)
```

```
            lout[x] = lin[x >> xshift];
        lout += c->width;
    }
    c->stride = c->width;
    free(c->pixels);
    c->pixels = out;
}

UJ_INLINE void ujConvert(ujContext *uj, unsigned char *pout) {
    int i;
    ujComponent* c;
    for (i = 0, c = uj->comp;  i < uj->ncomp;  ++i, ++c) {
        if (uj->fast_chroma) {
            ujUpsampleFast(uj, c);
            ujCheckError();
        } else {
            while ((c->width < uj->width) || (c->height < uj->height)) {
                if (c->width < uj->width) {
                    if (uj->co_sited_chroma) ujUpsampleHCoSited(c);
                                        else ujUpsampleHCentered(c);
                }
                ujCheckError();
                if (c->height < uj->height) {
                    if (uj->co_sited_chroma) ujUpsampleVCoSited(c);
                                        else ujUpsampleVCentered(c);
                }
                ujCheckError();
            }
        }
        if ((c->width < uj->width) || (c->height < uj->height)) ujThrow(UJ_INTERNAL_ERR
);
    }
    if (uj->ncomp == 3) {
        // convert to RGB
        int x, yy;
        const unsigned char *py  = uj->comp[0].pixels;
        const unsigned char *pcb = uj->comp[1].pixels;
        const unsigned char *pcr = uj->comp[2].pixels;
        for (yy = uj->height;  yy;  --yy) {
            for (x = 0;  x < uj->width;  ++x) {
                register int y = py[x] << 8;
                register int cb = pcb[x] - 128;
                register int cr = pcr[x] - 128;
                *pout++ = ujClip((y            + 359 * cr + 128) >> 8);
                *pout++ = ujClip((y -  88 * cb - 183 * cr + 128) >> 8);
                *pout++ = ujClip((y + 454 * cb            + 128) >> 8);
            }
            py += uj->comp[0].stride;
            pcb += uj->comp[1].stride;
            pcr += uj->comp[2].stride;
        }
    } else {
        // grayscale -> only remove stride
        unsigned char *pin = &uj->comp[0].pixels[uj->comp[0].stride];
        int y;
        for (y = uj->height - 1;  y;  --y) {
            memcpy(pout, pin, uj->width);
            pin += uj->comp[0].stride;
            pout += uj->width;
        }
    }
}
```

```
void ujDone(ujContext *uj) {
    int i;
    for (i = 0;  i < 3;  ++i)
        if (uj->comp[i].pixels)
            free((void*) uj->comp[i].pixels);
    if (uj->rgb)
        free((void*) uj->rgb);
}

void ujInit(ujContext *uj) {
    int save_no_decode = uj->no_decode;
    int save_fast_chroma = uj->fast_chroma;
    ujDone(uj);
    memset(uj, 0, sizeof(ujContext));
    uj->no_decode = save_no_decode;
    uj->fast_chroma = save_fast_chroma;
}

///////////////////////////////////////////////////////////////////////////////

UJ_INLINE unsigned short ujGetExif16(ujContext* uj, const unsigned char *p) {
    if (uj->exif_le)
        return p[0] + (p[1] << 8);
    else
        return (p[0] << 8) + p[1];
}

UJ_INLINE int ujGetExif32(ujContext* uj, const unsigned char *p) {
    if (uj->exif_le)
        return p[0] + (p[1] << 8) + (p[2] << 16) + (p[3] << 24);
    else
        return (p[0] << 24) + (p[1] << 16) + (p[2] << 8) + p[3];
}

UJ_INLINE void ujDecodeExif(ujContext* uj) {
    const unsigned char *ptr;
    int size, count, i;
    if (uj->no_decode || uj->fast_chroma) {
        ujSkipMarker(uj);
        return;
    }
    ujDecodeLength(uj);
    ptr = uj->pos;
    size = uj->length;
    ujSkip(uj, uj->length);
    if (size < 18) return;
    if (!memcmp(ptr, "Exif\0\0II*\0", 10))
        uj->exif_le = 1;
    else if (!memcmp(ptr, "Exif\0\0MM\0*", 10))
        uj->exif_le = 0;
    else
        return;  // invalid Exif header
    i = ujGetExif32(uj, ptr+10) + 6;
    if ((i < 14) || (i > (size - 2))) return;
    ptr += i;
    size -= i;
    count = ujGetExif16(uj, ptr);
    i = (size - 2) / 12;
    if (count > i) return;
    ptr += 2;
    while (count--) {
        if ((ujGetExif16(uj, ptr) == 0x0213) // tag = YCbCrPositioning
        &&  (ujGetExif16(uj, ptr + 2) == 3)  // type = SHORT
        &&  (ujGetExif32(uj, ptr + 4) == 1)  // length = 1
        ) {
            uj->co_sited_chroma = (ujGetExif16(uj, ptr + 8) == 2);
            return;
        }
        ptr += 12;
    }
}

////////////////////////////////////////////////////////////////////////////////

ujImage ujCreate(void) {
    ujContext *uj = (ujContext*) calloc(1, sizeof(ujContext));
    ujError = uj ? UJ_OK : UJ_OUT_OF_MEM;
    return (ujImage) uj;
}

void ujDisableDecoding(ujImage img) {
    ujContext *uj = (ujContext*) img;
    if (uj) {
        uj->no_decode = 1;
        ujError = UJ_OK;
    } else
        ujError = UJ_NO_CONTEXT;
}

void ujSetChromaMode(ujImage img, int mode) {
    ujContext *uj = (ujContext*) img;
    if (uj) {
        uj->fast_chroma = mode;
        ujError = UJ_OK;
    } else
        ujError = UJ_NO_CONTEXT;
}

ujImage ujDecode(ujImage img, const void* jpeg, const int size) {
    ujContext *uj = (ujContext*) (img ? img : ujCreate());
    if (img) ujInit(uj);
    ujError = UJ_OK;
    if (!uj)
        { ujError = UJ_OUT_OF_MEM; goto out; }
    uj->pos = (const unsigned char*) jpeg;
    uj->size = size & 0x7FFFFFFF;
    if (uj->size < 2)
        { ujError = UJ_NO_JPEG; goto out; }
    if ((uj->pos[0] ^ 0xFF) | (uj->pos[1] ^ 0xD8))
        { ujError = UJ_NO_JPEG; goto out; }
    ujSkip(uj, 2);
    while (!ujError) {
        if ((uj->size < 2) || (uj->pos[0] != 0xFF))
            { ujError = UJ_SYNTAX_ERROR; goto out; }
        ujSkip(uj, 2);
        switch (uj->pos[-1]) {
            case 0xC0: ujDecodeSOF(uj);  break;
            case 0xC4: ujDecodeDHT(uj);  break;
            case 0xDB: ujDecodeDQT(uj);  break;
            case 0xDD: ujDecodeDRI(uj);  break;
            case 0xDA: ujDecodeScan(uj); break;
            case 0xFE: ujSkipMarker(uj); break;
            case 0xE1: ujDecodeExif(uj); break;
            default:
                if ((uj->pos[-1] & 0xF0) == 0xE0)
                    ujSkipMarker(uj);
                else
                    { ujError = UJ_UNSUPPORTED; goto out; }
        }
    }
    if (ujError == __UJ_FINISHED) ujError = UJ_OK;
  out:
    if (ujError && !uj->valid) {
        if (!img)
            ujFree(uj);
        return NULL;
    }
    return (ujImage) uj;
}

ujImage ujDecodeFile(ujImage img, const char* filename) {
    FILE *f; size_t size;
    void *buf;
    ujError = UJ_OK;
    f = fopen(filename, "rb");
    if (!f) {
        ujError = UJ_IO_ERROR;
        return NULL;
    }
    fseek(f, 0, SEEK_END);
    size = ftell(f);
    fseek(f, 0, SEEK_SET);
#ifdef UJ_NODECODE_BLOCK_SIZE
    if (img && ((ujContext*)img)->no_decode && (size > UJ_NODECODE_BLOCK_SIZE))
        size = UJ_NODECODE_BLOCK_SIZE;
#endif
    buf = malloc(size);
    if (!buf) {
        fclose(f);
        ujError = UJ_OUT_OF_MEM;
        return NULL;
    }
    size = fread(buf, 1, size, f);
    fclose(f);
    img = ujDecode(img, buf, (int) size);
    free(buf);
    return img;
}

ujResult ujGetError(void) {
    return ujError;
}

int ujIsValid(ujImage img) {
    ujContext *uj = (ujContext*) img;
    if (!uj) { ujError = UJ_NO_CONTEXT; return 0; }
    return uj->valid;
}

int ujGetWidth(ujImage img) {
    ujContext *uj = (ujContext*) img;
    ujError = !uj ? UJ_NO_CONTEXT : (uj->valid ? UJ_OK : UJ_NOT_DECODED);
    return ujError ? 0 : uj->width;
}

int ujGetHeight(ujImage img) {
    ujContext *uj = (ujContext*) img;
    ujError = !uj ? UJ_NO_CONTEXT : (uj->valid ? UJ_OK : UJ_NOT_DECODED);
    return ujError ? 0 : uj->height;
```

```
}

int ujIsColor(ujImage img) {
    ujContext *uj = (ujContext*) img;
    ujError = !uj ? UJ_NO_CONTEXT : (uj->valid ? UJ_OK : UJ_NOT_DECODED);
    return ujError ? 0 : (uj->ncomp != 1);
}

int ujGetImageSize(ujImage img) {
    ujContext *uj = (ujContext*) img;
    ujError = !uj ? UJ_NO_CONTEXT : (uj->valid ? UJ_OK : UJ_NOT_DECODED);
    return ujError ? 0 : (uj->width * uj->height * uj->ncomp);
}

ujPlane* ujGetPlane(ujImage img, int num) {
    ujContext *uj = (ujContext*) img;
    ujError = !uj ? UJ_NO_CONTEXT : (uj->decoded ? UJ_OK : UJ_NOT_DECODED);
    if (!ujError && (num >= uj->ncomp)) ujError = UJ_INVALID_ARG;
    return ujError ? NULL : ((ujPlane*) &uj->comp[num]);
}

unsigned char* ujGetImage(ujImage img, unsigned char* dest) {
    ujContext *uj = (ujContext*) img;
    ujError = !uj ? UJ_NO_CONTEXT : (uj->decoded ? UJ_OK : UJ_NOT_DECODED);
    if (ujError) return NULL;
    if (dest) {
        if (uj->rgb)
            memcpy(dest, uj->rgb, uj->width * uj->height * uj->ncomp);
        else {
            ujConvert(uj, dest);
            if (ujError) return NULL;
        }
        return dest;
    } else {
        if (!uj->rgb) {
            uj->rgb = malloc(uj->width * uj->height * uj->ncomp);
            if (!uj->rgb) { ujError = UJ_OUT_OF_MEM; return NULL; }
            ujConvert(uj, uj->rgb);
            if (ujError) return NULL;
        }
        return uj->rgb;
    }
}

void ujDestroy(ujImage img) {
    ujError = UJ_OK;
    if (!img) { ujError = UJ_NO_CONTEXT; return; }
    ujDone((ujContext*) img);
    free(img);
}


 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
END OF FILE Î¼JPEG - C Source
NAME: ujpeg.c
DATE: 2018/01/12
TIME: 11:00:13
(C) 2018 WINDGO Inc.
```