/\* stb\_image\_write - v1.07 - public domain - http://nothings.org/stb/stb\_image\_write.h

writes out PNG/BMP/TGA/JPEG/HDR images to C stdio - Sean Barrett 2010-2015

no warranty implied; use at your own risk

Before #including,

#define STB\_IMAGE\_WRITE\_IMPLEMENTATION

in the file that you want to have the implementation.

Will probably not work correctly with strict-aliasing optimizations.

ABOUT:

This header file is a library for writing images to C stdio. It could be

adapted to write to memory or a general streaming interface; let me know.

The PNG output is not optimal; it is 20-50% larger than the file

written by a decent optimizing implementation. This library is designed

for source code compactness and simplicity, not optimal image file size

or run-time performance.

BUILDING:

You can #define STBIW\_ASSERT(x) before the #include to avoid using assert.h.

You can #define STBIW\_MALLOC(), STBIW\_REALLOC(), and STBIW\_FREE() to replace

malloc,realloc,free.

You can define STBIW\_MEMMOVE() to replace memmove()

USAGE:

There are four functions, one for each image file format:

int stbi\_write\_png(char const \*filename, int w, int h, int comp, const void \*data, int stride\_in\_bytes);

int stbi\_write\_bmp(char const \*filename, int w, int h, int comp, const void \*data);

int stbi\_write\_tga(char const \*filename, int w, int h, int comp, const void \*data);

int stbi\_write\_hdr(char const \*filename, int w, int h, int comp, const float \*data);

int stbi\_write\_jpg(char const \*filename, int w, int h, int comp, const float \*data);

There are also four equivalent functions that use an arbitrary write function. You are

expected to open/close your file-equivalent before and after calling these:

int stbi\_write\_png\_to\_func(stbi\_write\_func \*func, void \*context, int w, int h, int comp, const void \*data, int stride\_in\_bytes);

int stbi\_write\_bmp\_to\_func(stbi\_write\_func \*func, void \*context, int w, int h, int comp, const void \*data);

int stbi\_write\_tga\_to\_func(stbi\_write\_func \*func, void \*context, int w, int h, int comp, const void \*data);

int stbi\_write\_hdr\_to\_func(stbi\_write\_func \*func, void \*context, int w, int h, int comp, const float \*data);

int stbi\_write\_jpg\_to\_func(stbi\_write\_func \*func, void \*context, int x, int y, int comp, const void \*data, int quality);

where the callback is:

void stbi\_write\_func(void \*context, void \*data, int size);

You can define STBI\_WRITE\_NO\_STDIO to disable the file variant of these

functions, so the library will not use stdio.h at all. However, this will

also disable HDR writing, because it requires stdio for formatted output.

Each function returns 0 on failure and non-0 on success.

The functions create an image file defined by the parameters. The image

is a rectangle of pixels stored from left-to-right, top-to-bottom.

Each pixel contains 'comp' channels of data stored interleaved with 8-bits

per channel, in the following order: 1=Y, 2=YA, 3=RGB, 4=RGBA. (Y is

monochrome color.) The rectangle is 'w' pixels wide and 'h' pixels tall.

The \*data pointer points to the first byte of the top-left-most pixel.

For PNG, "stride\_in\_bytes" is the distance in bytes from the first byte of

a row of pixels to the first byte of the next row of pixels.

PNG creates output files with the same number of components as the input.

The BMP format expands Y to RGB in the file format and does not

output alpha.

PNG supports writing rectangles of data even when the bytes storing rows of

data are not consecutive in memory (e.g. sub-rectangles of a larger image),

by supplying the stride between the beginning of adjacent rows. The other

formats do not. (Thus you cannot write a native-format BMP through the BMP

writer, both because it is in BGR order and because it may have padding

at the end of the line.)

HDR expects linear float data. Since the format is always 32-bit rgb(e)

data, alpha (if provided) is discarded, and for monochrome data it is

replicated across all three channels.

TGA supports RLE or non-RLE compressed data. To use non-RLE-compressed

data, set the global variable 'stbi\_write\_tga\_with\_rle' to 0.

JPEG does ignore alpha channels in input data; quality is between 1 and 100.

Higher quality looks better but results in a bigger image.

JPEG baseline (no JPEG progressive).

CREDITS:

PNG/BMP/TGA

Sean Barrett

HDR

Baldur Karlsson

TGA monochrome:

Jean-Sebastien Guay

misc enhancements:

Tim Kelsey

TGA RLE

Alan Hickman

initial file IO callback implementation

Emmanuel Julien

JPEG

Jon Olick (original jo\_jpeg.cpp code)

Daniel Gibson

bugfixes:

github:Chribba

Guillaume Chereau

github:jry2

github:romigrou

Sergio Gonzalez

Jonas Karlsson

Filip Wasil

Thatcher Ulrich

github:poppolopoppo

Patrick Boettcher

LICENSE

See end of file for license information.

\*/

#ifndef INCLUDE\_STB\_IMAGE\_WRITE\_H

#define INCLUDE\_STB\_IMAGE\_WRITE\_H

#ifdef \_\_cplusplus

extern "C" {

#endif

#ifdef STB\_IMAGE\_WRITE\_STATIC

#define STBIWDEF static

#else

#define STBIWDEF extern

extern int stbi\_write\_tga\_with\_rle;

#endif

#ifndef STBI\_WRITE\_NO\_STDIO

STBIWDEF int stbi\_write\_png(char const \*filename, int w, int h, int comp, const void \*data, int stride\_in\_bytes);

STBIWDEF int stbi\_write\_bmp(char const \*filename, int w, int h, int comp, const void \*data);

STBIWDEF int stbi\_write\_tga(char const \*filename, int w, int h, int comp, const void \*data);

STBIWDEF int stbi\_write\_hdr(char const \*filename, int w, int h, int comp, const float \*data);

STBIWDEF int stbi\_write\_jpg(char const \*filename, int x, int y, int comp, const void \*data, int quality);

#endif

typedef void stbi\_write\_func(void \*context, void \*data, int size);

STBIWDEF int stbi\_write\_png\_to\_func(stbi\_write\_func \*func, void \*context, int w, int h, int comp, const void \*data, int stride\_in\_bytes);

STBIWDEF int stbi\_write\_bmp\_to\_func(stbi\_write\_func \*func, void \*context, int w, int h, int comp, const void \*data);

STBIWDEF int stbi\_write\_tga\_to\_func(stbi\_write\_func \*func, void \*context, int w, int h, int comp, const void \*data);

STBIWDEF int stbi\_write\_hdr\_to\_func(stbi\_write\_func \*func, void \*context, int w, int h, int comp, const float \*data);

STBIWDEF int stbi\_write\_jpg\_to\_func(stbi\_write\_func \*func, void \*context, int x, int y, int comp, const void \*data, int quality);

#ifdef \_\_cplusplus

}

#endif

#endif//INCLUDE\_STB\_IMAGE\_WRITE\_H

#ifdef STB\_IMAGE\_WRITE\_IMPLEMENTATION

#ifdef \_WIN32

#ifndef \_CRT\_SECURE\_NO\_WARNINGS

#define \_CRT\_SECURE\_NO\_WARNINGS

#endif

#ifndef \_CRT\_NONSTDC\_NO\_DEPRECATE

#define \_CRT\_NONSTDC\_NO\_DEPRECATE

#endif

#endif

#ifndef STBI\_WRITE\_NO\_STDIO

#include <stdio.h>

#endif // STBI\_WRITE\_NO\_STDIO

#include <stdarg.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#if defined(STBIW\_MALLOC) && defined(STBIW\_FREE) && (defined(STBIW\_REALLOC) || defined(STBIW\_REALLOC\_SIZED))

// ok

#elif !defined(STBIW\_MALLOC) && !defined(STBIW\_FREE) && !defined(STBIW\_REALLOC) && !defined(STBIW\_REALLOC\_SIZED)

// ok

#else

#error "Must define all or none of STBIW\_MALLOC, STBIW\_FREE, and STBIW\_REALLOC (or STBIW\_REALLOC\_SIZED)."

#endif

#ifndef STBIW\_MALLOC

#define STBIW\_MALLOC(sz) malloc(sz)

#define STBIW\_REALLOC(p,newsz) realloc(p,newsz)

#define STBIW\_FREE(p) free(p)

#endif

#ifndef STBIW\_REALLOC\_SIZED

#define STBIW\_REALLOC\_SIZED(p,oldsz,newsz) STBIW\_REALLOC(p,newsz)

#endif

#ifndef STBIW\_MEMMOVE

#define STBIW\_MEMMOVE(a,b,sz) memmove(a,b,sz)

#endif

#ifndef STBIW\_ASSERT

#include <assert.h>

#define STBIW\_ASSERT(x) assert(x)

#endif

#define STBIW\_UCHAR(x) (unsigned char) ((x) & 0xff)

typedef struct

{

stbi\_write\_func \*func;

void \*context;

} stbi\_\_write\_context;

// initialize a callback-based context

static void stbi\_\_start\_write\_callbacks(stbi\_\_write\_context \*s, stbi\_write\_func \*c, void \*context)

{

s->func = c;

s->context = context;

}

#ifndef STBI\_WRITE\_NO\_STDIO

static void stbi\_\_stdio\_write(void \*context, void \*data, int size)

{

fwrite(data,1,size,(FILE\*) context);

}

static int stbi\_\_start\_write\_file(stbi\_\_write\_context \*s, const char \*filename)

{

FILE \*f = fopen(filename, "wb");

stbi\_\_start\_write\_callbacks(s, stbi\_\_stdio\_write, (void \*) f);

return f != NULL;

}

static void stbi\_\_end\_write\_file(stbi\_\_write\_context \*s)

{

fclose((FILE \*)s->context);

}

#endif // !STBI\_WRITE\_NO\_STDIO

typedef unsigned int stbiw\_uint32;

typedef int stb\_image\_write\_test[sizeof(stbiw\_uint32)==4 ? 1 : -1];

#ifdef STB\_IMAGE\_WRITE\_STATIC

static int stbi\_write\_tga\_with\_rle = 1;

#else

int stbi\_write\_tga\_with\_rle = 1;

#endif

static void stbiw\_\_writefv(stbi\_\_write\_context \*s, const char \*fmt, va\_list v)

{

while (\*fmt) {

switch (\*fmt++) {

case ' ': break;

case '1': { unsigned char x = STBIW\_UCHAR(va\_arg(v, int));

s->func(s->context,&x,1);

break; }

case '2': { int x = va\_arg(v,int);

unsigned char b[2];

b[0] = STBIW\_UCHAR(x);

b[1] = STBIW\_UCHAR(x>>8);

s->func(s->context,b,2);

break; }

case '4': { stbiw\_uint32 x = va\_arg(v,int);

unsigned char b[4];

b[0]=STBIW\_UCHAR(x);

b[1]=STBIW\_UCHAR(x>>8);

b[2]=STBIW\_UCHAR(x>>16);

b[3]=STBIW\_UCHAR(x>>24);

s->func(s->context,b,4);

break; }

default:

STBIW\_ASSERT(0);

return;

}

}

}

static void stbiw\_\_writef(stbi\_\_write\_context \*s, const char \*fmt, ...)

{

va\_list v;

va\_start(v, fmt);

stbiw\_\_writefv(s, fmt, v);

va\_end(v);

}

static void stbiw\_\_putc(stbi\_\_write\_context \*s, unsigned char c)

{

s->func(s->context, &c, 1);

}

static void stbiw\_\_write3(stbi\_\_write\_context \*s, unsigned char a, unsigned char b, unsigned char c)

{

unsigned char arr[3];

arr[0] = a, arr[1] = b, arr[2] = c;

s->func(s->context, arr, 3);

}

static void stbiw\_\_write\_pixel(stbi\_\_write\_context \*s, int rgb\_dir, int comp, int write\_alpha, int expand\_mono, unsigned char \*d)

{

unsigned char bg[3] = { 255, 0, 255}, px[3];

int k;

if (write\_alpha < 0)

s->func(s->context, &d[comp - 1], 1);

switch (comp) {

case 2: // 2 pixels = mono + alpha, alpha is written separately, so same as 1-channel case

case 1:

if (expand\_mono)

stbiw\_\_write3(s, d[0], d[0], d[0]); // monochrome bmp

else

s->func(s->context, d, 1); // monochrome TGA

break;

case 4:

if (!write\_alpha) {

// composite against pink background

for (k = 0; k < 3; ++k)

px[k] = bg[k] + ((d[k] - bg[k]) \* d[3]) / 255;

stbiw\_\_write3(s, px[1 - rgb\_dir], px[1], px[1 + rgb\_dir]);

break;

}

/\* FALLTHROUGH \*/

case 3:

stbiw\_\_write3(s, d[1 - rgb\_dir], d[1], d[1 + rgb\_dir]);

break;

}

if (write\_alpha > 0)

s->func(s->context, &d[comp - 1], 1);

}

static void stbiw\_\_write\_pixels(stbi\_\_write\_context \*s, int rgb\_dir, int vdir, int x, int y, int comp, void \*data, int write\_alpha, int scanline\_pad, int expand\_mono)

{

stbiw\_uint32 zero = 0;

int i,j, j\_end;

if (y <= 0)

return;

if (vdir < 0)

j\_end = -1, j = y-1;

else

j\_end = y, j = 0;

for (; j != j\_end; j += vdir) {

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

unsigned char \*d = (unsigned char \*) data + (j\*x+i)\*comp;

stbiw\_\_write\_pixel(s, rgb\_dir, comp, write\_alpha, expand\_mono, d);

}

s->func(s->context, &zero, scanline\_pad);

}

}

static int stbiw\_\_outfile(stbi\_\_write\_context \*s, int rgb\_dir, int vdir, int x, int y, int comp, int expand\_mono, void \*data, int alpha, int pad, const char \*fmt, ...)

{

if (y < 0 || x < 0) {

return 0;

} else {

va\_list v;

va\_start(v, fmt);

stbiw\_\_writefv(s, fmt, v);

va\_end(v);

stbiw\_\_write\_pixels(s,rgb\_dir,vdir,x,y,comp,data,alpha,pad, expand\_mono);

return 1;

}

}

static int stbi\_write\_bmp\_core(stbi\_\_write\_context \*s, int x, int y, int comp, const void \*data)

{

int pad = (-x\*3) & 3;

return stbiw\_\_outfile(s,-1,-1,x,y,comp,1,(void \*) data,0,pad,

"11 4 22 4" "4 44 22 444444",

'B', 'M', 14+40+(x\*3+pad)\*y, 0,0, 14+40, // file header

40, x,y, 1,24, 0,0,0,0,0,0); // bitmap header

}

STBIWDEF int stbi\_write\_bmp\_to\_func(stbi\_write\_func \*func, void \*context, int x, int y, int comp, const void \*data)

{

stbi\_\_write\_context s;

stbi\_\_start\_write\_callbacks(&s, func, context);

return stbi\_write\_bmp\_core(&s, x, y, comp, data);

}

#ifndef STBI\_WRITE\_NO\_STDIO

STBIWDEF int stbi\_write\_bmp(char const \*filename, int x, int y, int comp, const void \*data)

{

stbi\_\_write\_context s;

if (stbi\_\_start\_write\_file(&s,filename)) {

int r = stbi\_write\_bmp\_core(&s, x, y, comp, data);

stbi\_\_end\_write\_file(&s);

return r;

} else

return 0;

}

#endif //!STBI\_WRITE\_NO\_STDIO

static int stbi\_write\_tga\_core(stbi\_\_write\_context \*s, int x, int y, int comp, void \*data)

{

int has\_alpha = (comp == 2 || comp == 4);

int colorbytes = has\_alpha ? comp-1 : comp;

int format = colorbytes < 2 ? 3 : 2; // 3 color channels (RGB/RGBA) = 2, 1 color channel (Y/YA) = 3

if (y < 0 || x < 0)

return 0;

if (!stbi\_write\_tga\_with\_rle) {

return stbiw\_\_outfile(s, -1, -1, x, y, comp, 0, (void \*) data, has\_alpha, 0,

"111 221 2222 11", 0, 0, format, 0, 0, 0, 0, 0, x, y, (colorbytes + has\_alpha) \* 8, has\_alpha \* 8);

} else {

int i,j,k;

stbiw\_\_writef(s, "111 221 2222 11", 0,0,format+8, 0,0,0, 0,0,x,y, (colorbytes + has\_alpha) \* 8, has\_alpha \* 8);

for (j = y - 1; j >= 0; --j) {

unsigned char \*row = (unsigned char \*) data + j \* x \* comp;

int len;

for (i = 0; i < x; i += len) {

unsigned char \*begin = row + i \* comp;

int diff = 1;

len = 1;

if (i < x - 1) {

++len;

diff = memcmp(begin, row + (i + 1) \* comp, comp);

if (diff) {

const unsigned char \*prev = begin;

for (k = i + 2; k < x && len < 128; ++k) {

if (memcmp(prev, row + k \* comp, comp)) {

prev += comp;

++len;

} else {

--len;

break;

}

}

} else {

for (k = i + 2; k < x && len < 128; ++k) {

if (!memcmp(begin, row + k \* comp, comp)) {

++len;

} else {

break;

}

}

}

}

if (diff) {

unsigned char header = STBIW\_UCHAR(len - 1);

s->func(s->context, &header, 1);

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

stbiw\_\_write\_pixel(s, -1, comp, has\_alpha, 0, begin + k \* comp);

}

} else {

unsigned char header = STBIW\_UCHAR(len - 129);

s->func(s->context, &header, 1);

stbiw\_\_write\_pixel(s, -1, comp, has\_alpha, 0, begin);

}

}

}

}

return 1;

}

STBIWDEF int stbi\_write\_tga\_to\_func(stbi\_write\_func \*func, void \*context, int x, int y, int comp, const void \*data)

{

stbi\_\_write\_context s;

stbi\_\_start\_write\_callbacks(&s, func, context);

return stbi\_write\_tga\_core(&s, x, y, comp, (void \*) data);

}

#ifndef STBI\_WRITE\_NO\_STDIO

STBIWDEF int stbi\_write\_tga(char const \*filename, int x, int y, int comp, const void \*data)

{

stbi\_\_write\_context s;

if (stbi\_\_start\_write\_file(&s,filename)) {

int r = stbi\_write\_tga\_core(&s, x, y, comp, (void \*) data);

stbi\_\_end\_write\_file(&s);

return r;

} else

return 0;

}

#endif

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

// Radiance RGBE HDR writer

// by Baldur Karlsson

#define stbiw\_\_max(a, b) ((a) > (b) ? (a) : (b))

void stbiw\_\_linear\_to\_rgbe(unsigned char \*rgbe, float \*linear)

{

int exponent;

float maxcomp = stbiw\_\_max(linear[0], stbiw\_\_max(linear[1], linear[2]));

if (maxcomp < 1e-32f) {

rgbe[0] = rgbe[1] = rgbe[2] = rgbe[3] = 0;

} else {

float normalize = (float) frexp(maxcomp, &exponent) \* 256.0f/maxcomp;

rgbe[0] = (unsigned char)(linear[0] \* normalize);

rgbe[1] = (unsigned char)(linear[1] \* normalize);

rgbe[2] = (unsigned char)(linear[2] \* normalize);

rgbe[3] = (unsigned char)(exponent + 128);

}

}

void stbiw\_\_write\_run\_data(stbi\_\_write\_context \*s, int length, unsigned char databyte)

{

unsigned char lengthbyte = STBIW\_UCHAR(length+128);

STBIW\_ASSERT(length+128 <= 255);

s->func(s->context, &lengthbyte, 1);

s->func(s->context, &databyte, 1);

}

void stbiw\_\_write\_dump\_data(stbi\_\_write\_context \*s, int length, unsigned char \*data)

{

unsigned char lengthbyte = STBIW\_UCHAR(length);

STBIW\_ASSERT(length <= 128); // inconsistent with spec but consistent with official code

s->func(s->context, &lengthbyte, 1);

s->func(s->context, data, length);

}

void stbiw\_\_write\_hdr\_scanline(stbi\_\_write\_context \*s, int width, int ncomp, unsigned char \*scratch, float \*scanline)

{

unsigned char scanlineheader[4] = { 2, 2, 0, 0 };

unsigned char rgbe[4];

float linear[3];

int x;

scanlineheader[2] = (width&0xff00)>>8;

scanlineheader[3] = (width&0x00ff);

/\* skip RLE for images too small or large \*/

if (width < 8 || width >= 32768) {

for (x=0; x < width; x++) {

switch (ncomp) {

case 4: /\* fallthrough \*/

case 3: linear[2] = scanline[x\*ncomp + 2];

linear[1] = scanline[x\*ncomp + 1];

linear[0] = scanline[x\*ncomp + 0];

break;

default:

linear[0] = linear[1] = linear[2] = scanline[x\*ncomp + 0];

break;

}

stbiw\_\_linear\_to\_rgbe(rgbe, linear);

s->func(s->context, rgbe, 4);

}

} else {

int c,r;

/\* encode into scratch buffer \*/

for (x=0; x < width; x++) {

switch(ncomp) {

case 4: /\* fallthrough \*/

case 3: linear[2] = scanline[x\*ncomp + 2];

linear[1] = scanline[x\*ncomp + 1];

linear[0] = scanline[x\*ncomp + 0];

break;

default:

linear[0] = linear[1] = linear[2] = scanline[x\*ncomp + 0];

break;

}

stbiw\_\_linear\_to\_rgbe(rgbe, linear);

scratch[x + width\*0] = rgbe[0];

scratch[x + width\*1] = rgbe[1];

scratch[x + width\*2] = rgbe[2];

scratch[x + width\*3] = rgbe[3];

}

s->func(s->context, scanlineheader, 4);

/\* RLE each component separately \*/

for (c=0; c < 4; c++) {

unsigned char \*comp = &scratch[width\*c];

x = 0;

while (x < width) {

// find first run

r = x;

while (r+2 < width) {

if (comp[r] == comp[r+1] && comp[r] == comp[r+2])

break;

++r;

}

if (r+2 >= width)

r = width;

// dump up to first run

while (x < r) {

int len = r-x;

if (len > 128) len = 128;

stbiw\_\_write\_dump\_data(s, len, &comp[x]);

x += len;

}

// if there's a run, output it

if (r+2 < width) { // same test as what we break out of in search loop, so only true if we break'd

// find next byte after run

while (r < width && comp[r] == comp[x])

++r;

// output run up to r

while (x < r) {

int len = r-x;

if (len > 127) len = 127;

stbiw\_\_write\_run\_data(s, len, comp[x]);

x += len;

}

}

}

}

}

}

static int stbi\_write\_hdr\_core(stbi\_\_write\_context \*s, int x, int y, int comp, float \*data)

{

if (y <= 0 || x <= 0 || data == NULL)

return 0;

else {

// Each component is stored separately. Allocate scratch space for full output scanline.

unsigned char \*scratch = (unsigned char \*) STBIW\_MALLOC(x\*4);

int i, len;

char buffer[128];

char header[] = "#?RADIANCE\n# Written by stb\_image\_write.h\nFORMAT=32-bit\_rle\_rgbe\n";

s->func(s->context, header, sizeof(header)-1);

len = sprintf(buffer, "EXPOSURE= 1.0000000000000\n\n-Y %d +X %d\n", y, x);

s->func(s->context, buffer, len);

for(i=0; i < y; i++)

stbiw\_\_write\_hdr\_scanline(s, x, comp, scratch, data + comp\*i\*x);

STBIW\_FREE(scratch);

return 1;

}

}

STBIWDEF int stbi\_write\_hdr\_to\_func(stbi\_write\_func \*func, void \*context, int x, int y, int comp, const float \*data)

{

stbi\_\_write\_context s;

stbi\_\_start\_write\_callbacks(&s, func, context);

return stbi\_write\_hdr\_core(&s, x, y, comp, (float \*) data);

}

#ifndef STBI\_WRITE\_NO\_STDIO

STBIWDEF int stbi\_write\_hdr(char const \*filename, int x, int y, int comp, const float \*data)

{

stbi\_\_write\_context s;

if (stbi\_\_start\_write\_file(&s,filename)) {

int r = stbi\_write\_hdr\_core(&s, x, y, comp, (float \*) data);

stbi\_\_end\_write\_file(&s);

return r;

} else

return 0;

}

#endif // STBI\_WRITE\_NO\_STDIO

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

//

// PNG writer

//

// stretchy buffer; stbiw\_\_sbpush() == vector<>::push\_back() -- stbiw\_\_sbcount() == vector<>::size()

#define stbiw\_\_sbraw(a) ((int \*) (a) - 2)

#define stbiw\_\_sbm(a) stbiw\_\_sbraw(a)[0]

#define stbiw\_\_sbn(a) stbiw\_\_sbraw(a)[1]

#define stbiw\_\_sbneedgrow(a,n) ((a)==0 || stbiw\_\_sbn(a)+n >= stbiw\_\_sbm(a))

#define stbiw\_\_sbmaybegrow(a,n) (stbiw\_\_sbneedgrow(a,(n)) ? stbiw\_\_sbgrow(a,n) : 0)

#define stbiw\_\_sbgrow(a,n) stbiw\_\_sbgrowf((void \*\*) &(a), (n), sizeof(\*(a)))

#define stbiw\_\_sbpush(a, v) (stbiw\_\_sbmaybegrow(a,1), (a)[stbiw\_\_sbn(a)++] = (v))

#define stbiw\_\_sbcount(a) ((a) ? stbiw\_\_sbn(a) : 0)

#define stbiw\_\_sbfree(a) ((a) ? STBIW\_FREE(stbiw\_\_sbraw(a)),0 : 0)

static void \*stbiw\_\_sbgrowf(void \*\*arr, int increment, int itemsize)

{

int m = \*arr ? 2\*stbiw\_\_sbm(\*arr)+increment : increment+1;

void \*p = STBIW\_REALLOC\_SIZED(\*arr ? stbiw\_\_sbraw(\*arr) : 0, \*arr ? (stbiw\_\_sbm(\*arr)\*itemsize + sizeof(int)\*2) : 0, itemsize \* m + sizeof(int)\*2);

STBIW\_ASSERT(p);

if (p) {

if (!\*arr) ((int \*) p)[1] = 0;

\*arr = (void \*) ((int \*) p + 2);

stbiw\_\_sbm(\*arr) = m;

}

return \*arr;

}

static unsigned char \*stbiw\_\_zlib\_flushf(unsigned char \*data, unsigned int \*bitbuffer, int \*bitcount)

{

while (\*bitcount >= 8) {

stbiw\_\_sbpush(data, STBIW\_UCHAR(\*bitbuffer));

\*bitbuffer >>= 8;

\*bitcount -= 8;

}

return data;

}

static int stbiw\_\_zlib\_bitrev(int code, int codebits)

{

int res=0;

while (codebits--) {

res = (res << 1) | (code & 1);

code >>= 1;

}

return res;

}

static unsigned int stbiw\_\_zlib\_countm(unsigned char \*a, unsigned char \*b, int limit)

{

int i;

for (i=0; i < limit && i < 258; ++i)

if (a[i] != b[i]) break;

return i;

}

static unsigned int stbiw\_\_zhash(unsigned char \*data)

{

stbiw\_uint32 hash = data[0] + (data[1] << 8) + (data[2] << 16);

hash ^= hash << 3;

hash += hash >> 5;

hash ^= hash << 4;

hash += hash >> 17;

hash ^= hash << 25;

hash += hash >> 6;

return hash;

}

#define stbiw\_\_zlib\_flush() (out = stbiw\_\_zlib\_flushf(out, &bitbuf, &bitcount))

#define stbiw\_\_zlib\_add(code,codebits) \

(bitbuf |= (code) << bitcount, bitcount += (codebits), stbiw\_\_zlib\_flush())

#define stbiw\_\_zlib\_huffa(b,c) stbiw\_\_zlib\_add(stbiw\_\_zlib\_bitrev(b,c),c)

// default huffman tables

#define stbiw\_\_zlib\_huff1(n) stbiw\_\_zlib\_huffa(0x30 + (n), 8)

#define stbiw\_\_zlib\_huff2(n) stbiw\_\_zlib\_huffa(0x190 + (n)-144, 9)

#define stbiw\_\_zlib\_huff3(n) stbiw\_\_zlib\_huffa(0 + (n)-256,7)

#define stbiw\_\_zlib\_huff4(n) stbiw\_\_zlib\_huffa(0xc0 + (n)-280,8)

#define stbiw\_\_zlib\_huff(n) ((n) <= 143 ? stbiw\_\_zlib\_huff1(n) : (n) <= 255 ? stbiw\_\_zlib\_huff2(n) : (n) <= 279 ? stbiw\_\_zlib\_huff3(n) : stbiw\_\_zlib\_huff4(n))

#define stbiw\_\_zlib\_huffb(n) ((n) <= 143 ? stbiw\_\_zlib\_huff1(n) : stbiw\_\_zlib\_huff2(n))

#define stbiw\_\_ZHASH 16384

unsigned char \* stbi\_zlib\_compress(unsigned char \*data, int data\_len, int \*out\_len, int quality)

{

static unsigned short lengthc[] = { 3,4,5,6,7,8,9,10,11,13,15,17,19,23,27,31,35,43,51,59,67,83,99,115,131,163,195,227,258, 259 };

static unsigned char lengtheb[]= { 0,0,0,0,0,0,0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 0 };

static unsigned short distc[] = { 1,2,3,4,5,7,9,13,17,25,33,49,65,97,129,193,257,385,513,769,1025,1537,2049,3073,4097,6145,8193,12289,16385,24577, 32768 };

static unsigned char disteb[] = { 0,0,0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,12,13,13 };

unsigned int bitbuf=0;

int i,j, bitcount=0;

unsigned char \*out = NULL;

unsigned char \*\*\*hash\_table = (unsigned char\*\*\*) STBIW\_MALLOC(stbiw\_\_ZHASH \* sizeof(char\*\*));

if (quality < 5) quality = 5;

stbiw\_\_sbpush(out, 0x78); // DEFLATE 32K window

stbiw\_\_sbpush(out, 0x5e); // FLEVEL = 1

stbiw\_\_zlib\_add(1,1); // BFINAL = 1

stbiw\_\_zlib\_add(1,2); // BTYPE = 1 -- fixed huffman

for (i=0; i < stbiw\_\_ZHASH; ++i)

hash\_table[i] = NULL;

i=0;

while (i < data\_len-3) {

// hash next 3 bytes of data to be compressed

int h = stbiw\_\_zhash(data+i)&(stbiw\_\_ZHASH-1), best=3;

unsigned char \*bestloc = 0;

unsigned char \*\*hlist = hash\_table[h];

int n = stbiw\_\_sbcount(hlist);

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

if (hlist[j]-data > i-32768) { // if entry lies within window

int d = stbiw\_\_zlib\_countm(hlist[j], data+i, data\_len-i);

if (d >= best) best=d,bestloc=hlist[j];

}

}

// when hash table entry is too long, delete half the entries

if (hash\_table[h] && stbiw\_\_sbn(hash\_table[h]) == 2\*quality) {

STBIW\_MEMMOVE(hash\_table[h], hash\_table[h]+quality, sizeof(hash\_table[h][0])\*quality);

stbiw\_\_sbn(hash\_table[h]) = quality;

}

stbiw\_\_sbpush(hash\_table[h],data+i);

if (bestloc) {

// "lazy matching" - check match at \*next\* byte, and if it's better, do cur byte as literal

h = stbiw\_\_zhash(data+i+1)&(stbiw\_\_ZHASH-1);

hlist = hash\_table[h];

n = stbiw\_\_sbcount(hlist);

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

if (hlist[j]-data > i-32767) {

int e = stbiw\_\_zlib\_countm(hlist[j], data+i+1, data\_len-i-1);

if (e > best) { // if next match is better, bail on current match

bestloc = NULL;

break;

}

}

}

}

if (bestloc) {

int d = (int) (data+i - bestloc); // distance back

STBIW\_ASSERT(d <= 32767 && best <= 258);

for (j=0; best > lengthc[j+1]-1; ++j);

stbiw\_\_zlib\_huff(j+257);

if (lengtheb[j]) stbiw\_\_zlib\_add(best - lengthc[j], lengtheb[j]);

for (j=0; d > distc[j+1]-1; ++j);

stbiw\_\_zlib\_add(stbiw\_\_zlib\_bitrev(j,5),5);

if (disteb[j]) stbiw\_\_zlib\_add(d - distc[j], disteb[j]);

i += best;

} else {

stbiw\_\_zlib\_huffb(data[i]);

++i;

}

}

// write out final bytes

for (;i < data\_len; ++i)

stbiw\_\_zlib\_huffb(data[i]);

stbiw\_\_zlib\_huff(256); // end of block

// pad with 0 bits to byte boundary

while (bitcount)

stbiw\_\_zlib\_add(0,1);

for (i=0; i < stbiw\_\_ZHASH; ++i)

(void) stbiw\_\_sbfree(hash\_table[i]);

STBIW\_FREE(hash\_table);

{

// compute adler32 on input

unsigned int s1=1, s2=0;

int blocklen = (int) (data\_len % 5552);

j=0;

while (j < data\_len) {

for (i=0; i < blocklen; ++i) s1 += data[j+i], s2 += s1;

s1 %= 65521, s2 %= 65521;

j += blocklen;

blocklen = 5552;

}

stbiw\_\_sbpush(out, STBIW\_UCHAR(s2 >> 8));

stbiw\_\_sbpush(out, STBIW\_UCHAR(s2));

stbiw\_\_sbpush(out, STBIW\_UCHAR(s1 >> 8));

stbiw\_\_sbpush(out, STBIW\_UCHAR(s1));

}

\*out\_len = stbiw\_\_sbn(out);

// make returned pointer freeable

STBIW\_MEMMOVE(stbiw\_\_sbraw(out), out, \*out\_len);

return (unsigned char \*) stbiw\_\_sbraw(out);

}

static unsigned int stbiw\_\_crc32(unsigned char \*buffer, int len)

{

static unsigned int crc\_table[256] =

{

0x00000000, 0x77073096, 0xEE0E612C, 0x990951BA, 0x076DC419, 0x706AF48F, 0xE963A535, 0x9E6495A3,

0x0eDB8832, 0x79DCB8A4, 0xE0D5E91E, 0x97D2D988, 0x09B64C2B, 0x7EB17CBD, 0xE7B82D07, 0x90BF1D91,

0x1DB71064, 0x6AB020F2, 0xF3B97148, 0x84BE41DE, 0x1ADAD47D, 0x6DDDE4EB, 0xF4D4B551, 0x83D385C7,

0x136C9856, 0x646BA8C0, 0xFD62F97A, 0x8A65C9EC, 0x14015C4F, 0x63066CD9, 0xFA0F3D63, 0x8D080DF5,

0x3B6E20C8, 0x4C69105E, 0xD56041E4, 0xA2677172, 0x3C03E4D1, 0x4B04D447, 0xD20D85FD, 0xA50AB56B,

0x35B5A8FA, 0x42B2986C, 0xDBBBC9D6, 0xACBCF940, 0x32D86CE3, 0x45DF5C75, 0xDCD60DCF, 0xABD13D59,

0x26D930AC, 0x51DE003A, 0xC8D75180, 0xBFD06116, 0x21B4F4B5, 0x56B3C423, 0xCFBA9599, 0xB8BDA50F,

0x2802B89E, 0x5F058808, 0xC60CD9B2, 0xB10BE924, 0x2F6F7C87, 0x58684C11, 0xC1611DAB, 0xB6662D3D,

0x76DC4190, 0x01DB7106, 0x98D220BC, 0xEFD5102A, 0x71B18589, 0x06B6B51F, 0x9FBFE4A5, 0xE8B8D433,

0x7807C9A2, 0x0F00F934, 0x9609A88E, 0xE10E9818, 0x7F6A0DBB, 0x086D3D2D, 0x91646C97, 0xE6635C01,

0x6B6B51F4, 0x1C6C6162, 0x856530D8, 0xF262004E, 0x6C0695ED, 0x1B01A57B, 0x8208F4C1, 0xF50FC457,

0x65B0D9C6, 0x12B7E950, 0x8BBEB8EA, 0xFCB9887C, 0x62DD1DDF, 0x15DA2D49, 0x8CD37CF3, 0xFBD44C65,

0x4DB26158, 0x3AB551CE, 0xA3BC0074, 0xD4BB30E2, 0x4ADFA541, 0x3DD895D7, 0xA4D1C46D, 0xD3D6F4FB,

0x4369E96A, 0x346ED9FC, 0xAD678846, 0xDA60B8D0, 0x44042D73, 0x33031DE5, 0xAA0A4C5F, 0xDD0D7CC9,

0x5005713C, 0x270241AA, 0xBE0B1010, 0xC90C2086, 0x5768B525, 0x206F85B3, 0xB966D409, 0xCE61E49F,

0x5EDEF90E, 0x29D9C998, 0xB0D09822, 0xC7D7A8B4, 0x59B33D17, 0x2EB40D81, 0xB7BD5C3B, 0xC0BA6CAD,

0xEDB88320, 0x9ABFB3B6, 0x03B6E20C, 0x74B1D29A, 0xEAD54739, 0x9DD277AF, 0x04DB2615, 0x73DC1683,

0xE3630B12, 0x94643B84, 0x0D6D6A3E, 0x7A6A5AA8, 0xE40ECF0B, 0x9309FF9D, 0x0A00AE27, 0x7D079EB1,

0xF00F9344, 0x8708A3D2, 0x1E01F268, 0x6906C2FE, 0xF762575D, 0x806567CB, 0x196C3671, 0x6E6B06E7,

0xFED41B76, 0x89D32BE0, 0x10DA7A5A, 0x67DD4ACC, 0xF9B9DF6F, 0x8EBEEFF9, 0x17B7BE43, 0x60B08ED5,

0xD6D6A3E8, 0xA1D1937E, 0x38D8C2C4, 0x4FDFF252, 0xD1BB67F1, 0xA6BC5767, 0x3FB506DD, 0x48B2364B,

0xD80D2BDA, 0xAF0A1B4C, 0x36034AF6, 0x41047A60, 0xDF60EFC3, 0xA867DF55, 0x316E8EEF, 0x4669BE79,

0xCB61B38C, 0xBC66831A, 0x256FD2A0, 0x5268E236, 0xCC0C7795, 0xBB0B4703, 0x220216B9, 0x5505262F,

0xC5BA3BBE, 0xB2BD0B28, 0x2BB45A92, 0x5CB36A04, 0xC2D7FFA7, 0xB5D0CF31, 0x2CD99E8B, 0x5BDEAE1D,

0x9B64C2B0, 0xEC63F226, 0x756AA39C, 0x026D930A, 0x9C0906A9, 0xEB0E363F, 0x72076785, 0x05005713,

0x95BF4A82, 0xE2B87A14, 0x7BB12BAE, 0x0CB61B38, 0x92D28E9B, 0xE5D5BE0D, 0x7CDCEFB7, 0x0BDBDF21,

0x86D3D2D4, 0xF1D4E242, 0x68DDB3F8, 0x1FDA836E, 0x81BE16CD, 0xF6B9265B, 0x6FB077E1, 0x18B74777,

0x88085AE6, 0xFF0F6A70, 0x66063BCA, 0x11010B5C, 0x8F659EFF, 0xF862AE69, 0x616BFFD3, 0x166CCF45,

0xA00AE278, 0xD70DD2EE, 0x4E048354, 0x3903B3C2, 0xA7672661, 0xD06016F7, 0x4969474D, 0x3E6E77DB,

0xAED16A4A, 0xD9D65ADC, 0x40DF0B66, 0x37D83BF0, 0xA9BCAE53, 0xDEBB9EC5, 0x47B2CF7F, 0x30B5FFE9,

0xBDBDF21C, 0xCABAC28A, 0x53B39330, 0x24B4A3A6, 0xBAD03605, 0xCDD70693, 0x54DE5729, 0x23D967BF,

0xB3667A2E, 0xC4614AB8, 0x5D681B02, 0x2A6F2B94, 0xB40BBE37, 0xC30C8EA1, 0x5A05DF1B, 0x2D02EF8D

};

unsigned int crc = ~0u;

int i;

for (i=0; i < len; ++i)

crc = (crc >> 8) ^ crc\_table[buffer[i] ^ (crc & 0xff)];

return ~crc;

}

#define stbiw\_\_wpng4(o,a,b,c,d) ((o)[0]=STBIW\_UCHAR(a),(o)[1]=STBIW\_UCHAR(b),(o)[2]=STBIW\_UCHAR(c),(o)[3]=STBIW\_UCHAR(d),(o)+=4)

#define stbiw\_\_wp32(data,v) stbiw\_\_wpng4(data, (v)>>24,(v)>>16,(v)>>8,(v));

#define stbiw\_\_wptag(data,s) stbiw\_\_wpng4(data, s[0],s[1],s[2],s[3])

static void stbiw\_\_wpcrc(unsigned char \*\*data, int len)

{

unsigned int crc = stbiw\_\_crc32(\*data - len - 4, len+4);

stbiw\_\_wp32(\*data, crc);

}

static unsigned char stbiw\_\_paeth(int a, int b, int c)

{

int p = a + b - c, pa = abs(p-a), pb = abs(p-b), pc = abs(p-c);

if (pa <= pb && pa <= pc) return STBIW\_UCHAR(a);

if (pb <= pc) return STBIW\_UCHAR(b);

return STBIW\_UCHAR(c);

}

// @OPTIMIZE: provide an option that always forces left-predict or paeth predict

unsigned char \*stbi\_write\_png\_to\_mem(unsigned char \*pixels, int stride\_bytes, int x, int y, int n, int \*out\_len)

{

int ctype[5] = { -1, 0, 4, 2, 6 };

unsigned char sig[8] = { 137,80,78,71,13,10,26,10 };

unsigned char \*out,\*o, \*filt, \*zlib;

signed char \*line\_buffer;

int i,j,k,p,zlen;

if (stride\_bytes == 0)

stride\_bytes = x \* n;

filt = (unsigned char \*) STBIW\_MALLOC((x\*n+1) \* y); if (!filt) return 0;

line\_buffer = (signed char \*) STBIW\_MALLOC(x \* n); if (!line\_buffer) { STBIW\_FREE(filt); return 0; }

for (j=0; j < y; ++j) {

static int mapping[] = { 0,1,2,3,4 };

static int firstmap[] = { 0,1,0,5,6 };

int \*mymap = (j != 0) ? mapping : firstmap;

int best = 0, bestval = 0x7fffffff;

for (p=0; p < 2; ++p) {

for (k= p?best:0; k < 5; ++k) { // @TODO: clarity: rewrite this to go 0..5, and 'continue' the unwanted ones during 2nd pass

int type = mymap[k],est=0;

unsigned char \*z = pixels + stride\_bytes\*j;

for (i=0; i < n; ++i)

switch (type) {

case 0: line\_buffer[i] = z[i]; break;

case 1: line\_buffer[i] = z[i]; break;

case 2: line\_buffer[i] = z[i] - z[i-stride\_bytes]; break;

case 3: line\_buffer[i] = z[i] - (z[i-stride\_bytes]>>1); break;

case 4: line\_buffer[i] = (signed char) (z[i] - stbiw\_\_paeth(0,z[i-stride\_bytes],0)); break;

case 5: line\_buffer[i] = z[i]; break;

case 6: line\_buffer[i] = z[i]; break;

}

for (i=n; i < x\*n; ++i) {

switch (type) {

case 0: line\_buffer[i] = z[i]; break;

case 1: line\_buffer[i] = z[i] - z[i-n]; break;

case 2: line\_buffer[i] = z[i] - z[i-stride\_bytes]; break;

case 3: line\_buffer[i] = z[i] - ((z[i-n] + z[i-stride\_bytes])>>1); break;

case 4: line\_buffer[i] = z[i] - stbiw\_\_paeth(z[i-n], z[i-stride\_bytes], z[i-stride\_bytes-n]); break;

case 5: line\_buffer[i] = z[i] - (z[i-n]>>1); break;

case 6: line\_buffer[i] = z[i] - stbiw\_\_paeth(z[i-n], 0,0); break;

}

}

if (p) break;

for (i=0; i < x\*n; ++i)

est += abs((signed char) line\_buffer[i]);

if (est < bestval) { bestval = est; best = k; }

}

}

// when we get here, best contains the filter type, and line\_buffer contains the data

filt[j\*(x\*n+1)] = (unsigned char) best;

STBIW\_MEMMOVE(filt+j\*(x\*n+1)+1, line\_buffer, x\*n);

}

STBIW\_FREE(line\_buffer);

zlib = stbi\_zlib\_compress(filt, y\*( x\*n+1), &zlen, 8); // increase 8 to get smaller but use more memory

STBIW\_FREE(filt);

if (!zlib) return 0;

// each tag requires 12 bytes of overhead

out = (unsigned char \*) STBIW\_MALLOC(8 + 12+13 + 12+zlen + 12);

if (!out) return 0;

\*out\_len = 8 + 12+13 + 12+zlen + 12;

o=out;

STBIW\_MEMMOVE(o,sig,8); o+= 8;

stbiw\_\_wp32(o, 13); // header length

stbiw\_\_wptag(o, "IHDR");

stbiw\_\_wp32(o, x);

stbiw\_\_wp32(o, y);

\*o++ = 8;

\*o++ = STBIW\_UCHAR(ctype[n]);

\*o++ = 0;

\*o++ = 0;

\*o++ = 0;

stbiw\_\_wpcrc(&o,13);

stbiw\_\_wp32(o, zlen);

stbiw\_\_wptag(o, "IDAT");

STBIW\_MEMMOVE(o, zlib, zlen);

o += zlen;

STBIW\_FREE(zlib);

stbiw\_\_wpcrc(&o, zlen);

stbiw\_\_wp32(o,0);

stbiw\_\_wptag(o, "IEND");

stbiw\_\_wpcrc(&o,0);

STBIW\_ASSERT(o == out + \*out\_len);

return out;

}

#ifndef STBI\_WRITE\_NO\_STDIO

STBIWDEF int stbi\_write\_png(char const \*filename, int x, int y, int comp, const void \*data, int stride\_bytes)

{

FILE \*f;

int len;

unsigned char \*png = stbi\_write\_png\_to\_mem((unsigned char \*) data, stride\_bytes, x, y, comp, &len);

if (png == NULL) return 0;

f = fopen(filename, "wb");

if (!f) { STBIW\_FREE(png); return 0; }

fwrite(png, 1, len, f);

fclose(f);

STBIW\_FREE(png);

return 1;

}

#endif

STBIWDEF int stbi\_write\_png\_to\_func(stbi\_write\_func \*func, void \*context, int x, int y, int comp, const void \*data, int stride\_bytes)

{

int len;

unsigned char \*png = stbi\_write\_png\_to\_mem((unsigned char \*) data, stride\_bytes, x, y, comp, &len);

if (png == NULL) return 0;

func(context, png, len);

STBIW\_FREE(png);

return 1;

}

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

\*

\* JPEG writer

\*

\* This is based on Jon Olick's jo\_jpeg.cpp:

\* public domain Simple, Minimalistic JPEG writer - http://www.jonolick.com/code.html

\*/

static const unsigned char stbiw\_\_jpg\_ZigZag[] = { 0,1,5,6,14,15,27,28,2,4,7,13,16,26,29,42,3,8,12,17,25,30,41,43,9,11,18,

24,31,40,44,53,10,19,23,32,39,45,52,54,20,22,33,38,46,51,55,60,21,34,37,47,50,56,59,61,35,36,48,49,57,58,62,63 };

static void stbiw\_\_jpg\_writeBits(stbi\_\_write\_context \*s, int \*bitBufP, int \*bitCntP, const unsigned short \*bs) {

int bitBuf = \*bitBufP, bitCnt = \*bitCntP;

bitCnt += bs[1];

bitBuf |= bs[0] << (24 - bitCnt);

while(bitCnt >= 8) {

unsigned char c = (bitBuf >> 16) & 255;

stbiw\_\_putc(s, c);

if(c == 255) {

stbiw\_\_putc(s, 0);

}

bitBuf <<= 8;

bitCnt -= 8;

}

\*bitBufP = bitBuf;

\*bitCntP = bitCnt;

}

static void stbiw\_\_jpg\_DCT(float \*d0p, float \*d1p, float \*d2p, float \*d3p, float \*d4p, float \*d5p, float \*d6p, float \*d7p) {

float d0 = \*d0p, d1 = \*d1p, d2 = \*d2p, d3 = \*d3p, d4 = \*d4p, d5 = \*d5p, d6 = \*d6p, d7 = \*d7p;

float z1, z2, z3, z4, z5, z11, z13;

float tmp0 = d0 + d7;

float tmp7 = d0 - d7;

float tmp1 = d1 + d6;

float tmp6 = d1 - d6;

float tmp2 = d2 + d5;

float tmp5 = d2 - d5;

float tmp3 = d3 + d4;

float tmp4 = d3 - d4;

// Even part

float tmp10 = tmp0 + tmp3; // phase 2

float tmp13 = tmp0 - tmp3;

float tmp11 = tmp1 + tmp2;

float tmp12 = tmp1 - tmp2;

d0 = tmp10 + tmp11; // phase 3

d4 = tmp10 - tmp11;

z1 = (tmp12 + tmp13) \* 0.707106781f; // c4

d2 = tmp13 + z1; // phase 5

d6 = tmp13 - z1;

// Odd part

tmp10 = tmp4 + tmp5; // phase 2

tmp11 = tmp5 + tmp6;

tmp12 = tmp6 + tmp7;

// The rotator is modified from fig 4-8 to avoid extra negations.

z5 = (tmp10 - tmp12) \* 0.382683433f; // c6

z2 = tmp10 \* 0.541196100f + z5; // c2-c6

z4 = tmp12 \* 1.306562965f + z5; // c2+c6

z3 = tmp11 \* 0.707106781f; // c4

z11 = tmp7 + z3; // phase 5

z13 = tmp7 - z3;

\*d5p = z13 + z2; // phase 6

\*d3p = z13 - z2;

\*d1p = z11 + z4;

\*d7p = z11 - z4;

\*d0p = d0; \*d2p = d2; \*d4p = d4; \*d6p = d6;

}

static void stbiw\_\_jpg\_calcBits(int val, unsigned short bits[2]) {

int tmp1 = val < 0 ? -val : val;

val = val < 0 ? val-1 : val;

bits[1] = 1;

while(tmp1 >>= 1) {

++bits[1];

}

bits[0] = val & ((1<<bits[1])-1);

}

static int stbiw\_\_jpg\_processDU(stbi\_\_write\_context \*s, int \*bitBuf, int \*bitCnt, float \*CDU, float \*fdtbl, int DC, const unsigned short HTDC[256][2], const unsigned short HTAC[256][2]) {

const unsigned short EOB[2] = { HTAC[0x00][0], HTAC[0x00][1] };

const unsigned short M16zeroes[2] = { HTAC[0xF0][0], HTAC[0xF0][1] };

int dataOff, i, diff, end0pos;

int DU[64];

// DCT rows

for(dataOff=0; dataOff<64; dataOff+=8) {

stbiw\_\_jpg\_DCT(&CDU[dataOff], &CDU[dataOff+1], &CDU[dataOff+2], &CDU[dataOff+3], &CDU[dataOff+4], &CDU[dataOff+5], &CDU[dataOff+6], &CDU[dataOff+7]);

}

// DCT columns

for(dataOff=0; dataOff<8; ++dataOff) {

stbiw\_\_jpg\_DCT(&CDU[dataOff], &CDU[dataOff+8], &CDU[dataOff+16], &CDU[dataOff+24], &CDU[dataOff+32], &CDU[dataOff+40], &CDU[dataOff+48], &CDU[dataOff+56]);

}

// Quantize/descale/zigzag the coefficients

for(i=0; i<64; ++i) {

float v = CDU[i]\*fdtbl[i];

// DU[stbiw\_\_jpg\_ZigZag[i]] = (int)(v < 0 ? ceilf(v - 0.5f) : floorf(v + 0.5f));

// ceilf() and floorf() are C99, not C89, but I /think/ they're not needed here anyway?

DU[stbiw\_\_jpg\_ZigZag[i]] = (int)(v < 0 ? v - 0.5f : v + 0.5f);

}

// Encode DC

diff = DU[0] - DC;

if (diff == 0) {

stbiw\_\_jpg\_writeBits(s, bitBuf, bitCnt, HTDC[0]);

} else {

unsigned short bits[2];

stbiw\_\_jpg\_calcBits(diff, bits);

stbiw\_\_jpg\_writeBits(s, bitBuf, bitCnt, HTDC[bits[1]]);

stbiw\_\_jpg\_writeBits(s, bitBuf, bitCnt, bits);

}

// Encode ACs

end0pos = 63;

for(; (end0pos>0)&&(DU[end0pos]==0); --end0pos) {

}

// end0pos = first element in reverse order !=0

if(end0pos == 0) {

stbiw\_\_jpg\_writeBits(s, bitBuf, bitCnt, EOB);

return DU[0];

}

for(i = 1; i <= end0pos; ++i) {

int startpos = i;

int nrzeroes;

unsigned short bits[2];

for (; DU[i]==0 && i<=end0pos; ++i) {

}

nrzeroes = i-startpos;

if ( nrzeroes >= 16 ) {

int lng = nrzeroes>>4;

int nrmarker;

for (nrmarker=1; nrmarker <= lng; ++nrmarker)

stbiw\_\_jpg\_writeBits(s, bitBuf, bitCnt, M16zeroes);

nrzeroes &= 15;

}

stbiw\_\_jpg\_calcBits(DU[i], bits);

stbiw\_\_jpg\_writeBits(s, bitBuf, bitCnt, HTAC[(nrzeroes<<4)+bits[1]]);

stbiw\_\_jpg\_writeBits(s, bitBuf, bitCnt, bits);

}

if(end0pos != 63) {

stbiw\_\_jpg\_writeBits(s, bitBuf, bitCnt, EOB);

}

return DU[0];

}

static int stbi\_write\_jpg\_core(stbi\_\_write\_context \*s, int width, int height, int comp, const void\* data, int quality) {

// Constants that don't pollute global namespace

static const unsigned char std\_dc\_luminance\_nrcodes[] = {0,0,1,5,1,1,1,1,1,1,0,0,0,0,0,0,0};

static const unsigned char std\_dc\_luminance\_values[] = {0,1,2,3,4,5,6,7,8,9,10,11};

static const unsigned char std\_ac\_luminance\_nrcodes[] = {0,0,2,1,3,3,2,4,3,5,5,4,4,0,0,1,0x7d};

static const unsigned char std\_ac\_luminance\_values[] = {

0x01,0x02,0x03,0x00,0x04,0x11,0x05,0x12,0x21,0x31,0x41,0x06,0x13,0x51,0x61,0x07,0x22,0x71,0x14,0x32,0x81,0x91,0xa1,0x08,

0x23,0x42,0xb1,0xc1,0x15,0x52,0xd1,0xf0,0x24,0x33,0x62,0x72,0x82,0x09,0x0a,0x16,0x17,0x18,0x19,0x1a,0x25,0x26,0x27,0x28,

0x29,0x2a,0x34,0x35,0x36,0x37,0x38,0x39,0x3a,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x53,0x54,0x55,0x56,0x57,0x58,0x59,

0x5a,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6a,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7a,0x83,0x84,0x85,0x86,0x87,0x88,0x89,

0x8a,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xb2,0xb3,0xb4,0xb5,0xb6,

0xb7,0xb8,0xb9,0xba,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xca,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,0xe1,0xe2,

0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa

};

static const unsigned char std\_dc\_chrominance\_nrcodes[] = {0,0,3,1,1,1,1,1,1,1,1,1,0,0,0,0,0};

static const unsigned char std\_dc\_chrominance\_values[] = {0,1,2,3,4,5,6,7,8,9,10,11};

static const unsigned char std\_ac\_chrominance\_nrcodes[] = {0,0,2,1,2,4,4,3,4,7,5,4,4,0,1,2,0x77};

static const unsigned char std\_ac\_chrominance\_values[] = {

0x00,0x01,0x02,0x03,0x11,0x04,0x05,0x21,0x31,0x06,0x12,0x41,0x51,0x07,0x61,0x71,0x13,0x22,0x32,0x81,0x08,0x14,0x42,0x91,

0xa1,0xb1,0xc1,0x09,0x23,0x33,0x52,0xf0,0x15,0x62,0x72,0xd1,0x0a,0x16,0x24,0x34,0xe1,0x25,0xf1,0x17,0x18,0x19,0x1a,0x26,

0x27,0x28,0x29,0x2a,0x35,0x36,0x37,0x38,0x39,0x3a,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x53,0x54,0x55,0x56,0x57,0x58,

0x59,0x5a,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6a,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7a,0x82,0x83,0x84,0x85,0x86,0x87,

0x88,0x89,0x8a,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xb2,0xb3,0xb4,

0xb5,0xb6,0xb7,0xb8,0xb9,0xba,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xca,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,

0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa

};

// Huffman tables

static const unsigned short YDC\_HT[256][2] = { {0,2},{2,3},{3,3},{4,3},{5,3},{6,3},{14,4},{30,5},{62,6},{126,7},{254,8},{510,9}};

static const unsigned short UVDC\_HT[256][2] = { {0,2},{1,2},{2,2},{6,3},{14,4},{30,5},{62,6},{126,7},{254,8},{510,9},{1022,10},{2046,11}};

static const unsigned short YAC\_HT[256][2] = {

{10,4},{0,2},{1,2},{4,3},{11,4},{26,5},{120,7},{248,8},{1014,10},{65410,16},{65411,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{12,4},{27,5},{121,7},{502,9},{2038,11},{65412,16},{65413,16},{65414,16},{65415,16},{65416,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{28,5},{249,8},{1015,10},{4084,12},{65417,16},{65418,16},{65419,16},{65420,16},{65421,16},{65422,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{58,6},{503,9},{4085,12},{65423,16},{65424,16},{65425,16},{65426,16},{65427,16},{65428,16},{65429,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{59,6},{1016,10},{65430,16},{65431,16},{65432,16},{65433,16},{65434,16},{65435,16},{65436,16},{65437,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{122,7},{2039,11},{65438,16},{65439,16},{65440,16},{65441,16},{65442,16},{65443,16},{65444,16},{65445,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{123,7},{4086,12},{65446,16},{65447,16},{65448,16},{65449,16},{65450,16},{65451,16},{65452,16},{65453,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{250,8},{4087,12},{65454,16},{65455,16},{65456,16},{65457,16},{65458,16},{65459,16},{65460,16},{65461,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{504,9},{32704,15},{65462,16},{65463,16},{65464,16},{65465,16},{65466,16},{65467,16},{65468,16},{65469,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{505,9},{65470,16},{65471,16},{65472,16},{65473,16},{65474,16},{65475,16},{65476,16},{65477,16},{65478,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{506,9},{65479,16},{65480,16},{65481,16},{65482,16},{65483,16},{65484,16},{65485,16},{65486,16},{65487,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{1017,10},{65488,16},{65489,16},{65490,16},{65491,16},{65492,16},{65493,16},{65494,16},{65495,16},{65496,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{1018,10},{65497,16},{65498,16},{65499,16},{65500,16},{65501,16},{65502,16},{65503,16},{65504,16},{65505,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{2040,11},{65506,16},{65507,16},{65508,16},{65509,16},{65510,16},{65511,16},{65512,16},{65513,16},{65514,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{65515,16},{65516,16},{65517,16},{65518,16},{65519,16},{65520,16},{65521,16},{65522,16},{65523,16},{65524,16},{0,0},{0,0},{0,0},{0,0},{0,0},

{2041,11},{65525,16},{65526,16},{65527,16},{65528,16},{65529,16},{65530,16},{65531,16},{65532,16},{65533,16},{65534,16},{0,0},{0,0},{0,0},{0,0},{0,0}

};

static const unsigned short UVAC\_HT[256][2] = {

{0,2},{1,2},{4,3},{10,4},{24,5},{25,5},{56,6},{120,7},{500,9},{1014,10},{4084,12},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{11,4},{57,6},{246,8},{501,9},{2038,11},{4085,12},{65416,16},{65417,16},{65418,16},{65419,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{26,5},{247,8},{1015,10},{4086,12},{32706,15},{65420,16},{65421,16},{65422,16},{65423,16},{65424,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{27,5},{248,8},{1016,10},{4087,12},{65425,16},{65426,16},{65427,16},{65428,16},{65429,16},{65430,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{58,6},{502,9},{65431,16},{65432,16},{65433,16},{65434,16},{65435,16},{65436,16},{65437,16},{65438,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{59,6},{1017,10},{65439,16},{65440,16},{65441,16},{65442,16},{65443,16},{65444,16},{65445,16},{65446,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{121,7},{2039,11},{65447,16},{65448,16},{65449,16},{65450,16},{65451,16},{65452,16},{65453,16},{65454,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{122,7},{2040,11},{65455,16},{65456,16},{65457,16},{65458,16},{65459,16},{65460,16},{65461,16},{65462,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{249,8},{65463,16},{65464,16},{65465,16},{65466,16},{65467,16},{65468,16},{65469,16},{65470,16},{65471,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{503,9},{65472,16},{65473,16},{65474,16},{65475,16},{65476,16},{65477,16},{65478,16},{65479,16},{65480,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{504,9},{65481,16},{65482,16},{65483,16},{65484,16},{65485,16},{65486,16},{65487,16},{65488,16},{65489,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{505,9},{65490,16},{65491,16},{65492,16},{65493,16},{65494,16},{65495,16},{65496,16},{65497,16},{65498,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{506,9},{65499,16},{65500,16},{65501,16},{65502,16},{65503,16},{65504,16},{65505,16},{65506,16},{65507,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{2041,11},{65508,16},{65509,16},{65510,16},{65511,16},{65512,16},{65513,16},{65514,16},{65515,16},{65516,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},

{16352,14},{65517,16},{65518,16},{65519,16},{65520,16},{65521,16},{65522,16},{65523,16},{65524,16},{65525,16},{0,0},{0,0},{0,0},{0,0},{0,0},

{1018,10},{32707,15},{65526,16},{65527,16},{65528,16},{65529,16},{65530,16},{65531,16},{65532,16},{65533,16},{65534,16},{0,0},{0,0},{0,0},{0,0},{0,0}

};

static const int YQT[] = {16,11,10,16,24,40,51,61,12,12,14,19,26,58,60,55,14,13,16,24,40,57,69,56,14,17,22,29,51,87,80,62,18,22,

37,56,68,109,103,77,24,35,55,64,81,104,113,92,49,64,78,87,103,121,120,101,72,92,95,98,112,100,103,99};

static const int UVQT[] = {17,18,24,47,99,99,99,99,18,21,26,66,99,99,99,99,24,26,56,99,99,99,99,99,47,66,99,99,99,99,99,99,

99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99};

static const float aasf[] = { 1.0f \* 2.828427125f, 1.387039845f \* 2.828427125f, 1.306562965f \* 2.828427125f, 1.175875602f \* 2.828427125f,

1.0f \* 2.828427125f, 0.785694958f \* 2.828427125f, 0.541196100f \* 2.828427125f, 0.275899379f \* 2.828427125f };

int row, col, i, k;

float fdtbl\_Y[64], fdtbl\_UV[64];

unsigned char YTable[64], UVTable[64];

if(!data || !width || !height || comp > 4 || comp < 1) {

return 0;

}

quality = quality ? quality : 90;

quality = quality < 1 ? 1 : quality > 100 ? 100 : quality;

quality = quality < 50 ? 5000 / quality : 200 - quality \* 2;

for(i = 0; i < 64; ++i) {

int uvti, yti = (YQT[i]\*quality+50)/100;

YTable[stbiw\_\_jpg\_ZigZag[i]] = (unsigned char) (yti < 1 ? 1 : yti > 255 ? 255 : yti);

uvti = (UVQT[i]\*quality+50)/100;

UVTable[stbiw\_\_jpg\_ZigZag[i]] = (unsigned char) (uvti < 1 ? 1 : uvti > 255 ? 255 : uvti);

}

for(row = 0, k = 0; row < 8; ++row) {

for(col = 0; col < 8; ++col, ++k) {

fdtbl\_Y[k] = 1 / (YTable [stbiw\_\_jpg\_ZigZag[k]] \* aasf[row] \* aasf[col]);

fdtbl\_UV[k] = 1 / (UVTable[stbiw\_\_jpg\_ZigZag[k]] \* aasf[row] \* aasf[col]);

}

}

// Write Headers

{

static const unsigned char head0[] = { 0xFF,0xD8,0xFF,0xE0,0,0x10,'J','F','I','F',0,1,1,0,0,1,0,1,0,0,0xFF,0xDB,0,0x84,0 };

static const unsigned char head2[] = { 0xFF,0xDA,0,0xC,3,1,0,2,0x11,3,0x11,0,0x3F,0 };

const unsigned char head1[] = { 0xFF,0xC0,0,0x11,8,(unsigned char)(height>>8),STBIW\_UCHAR(height),(unsigned char)(width>>8),STBIW\_UCHAR(width),

3,1,0x11,0,2,0x11,1,3,0x11,1,0xFF,0xC4,0x01,0xA2,0 };

s->func(s->context, (void\*)head0, sizeof(head0));

s->func(s->context, (void\*)YTable, sizeof(YTable));

stbiw\_\_putc(s, 1);

s->func(s->context, UVTable, sizeof(UVTable));

s->func(s->context, (void\*)head1, sizeof(head1));

s->func(s->context, (void\*)(std\_dc\_luminance\_nrcodes+1), sizeof(std\_dc\_luminance\_nrcodes)-1);

s->func(s->context, (void\*)std\_dc\_luminance\_values, sizeof(std\_dc\_luminance\_values));

stbiw\_\_putc(s, 0x10); // HTYACinfo

s->func(s->context, (void\*)(std\_ac\_luminance\_nrcodes+1), sizeof(std\_ac\_luminance\_nrcodes)-1);

s->func(s->context, (void\*)std\_ac\_luminance\_values, sizeof(std\_ac\_luminance\_values));

stbiw\_\_putc(s, 1); // HTUDCinfo

s->func(s->context, (void\*)(std\_dc\_chrominance\_nrcodes+1), sizeof(std\_dc\_chrominance\_nrcodes)-1);

s->func(s->context, (void\*)std\_dc\_chrominance\_values, sizeof(std\_dc\_chrominance\_values));

stbiw\_\_putc(s, 0x11); // HTUACinfo

s->func(s->context, (void\*)(std\_ac\_chrominance\_nrcodes+1), sizeof(std\_ac\_chrominance\_nrcodes)-1);

s->func(s->context, (void\*)std\_ac\_chrominance\_values, sizeof(std\_ac\_chrominance\_values));

s->func(s->context, (void\*)head2, sizeof(head2));

}

// Encode 8x8 macroblocks

{

static const unsigned short fillBits[] = {0x7F, 7};

const unsigned char \*imageData = (const unsigned char \*)data;

int DCY=0, DCU=0, DCV=0;

int bitBuf=0, bitCnt=0;

// comp == 2 is grey+alpha (alpha is ignored)

int ofsG = comp > 2 ? 1 : 0, ofsB = comp > 2 ? 2 : 0;

int x, y, pos;

for(y = 0; y < height; y += 8) {

for(x = 0; x < width; x += 8) {

float YDU[64], UDU[64], VDU[64];

for(row = y, pos = 0; row < y+8; ++row) {

for(col = x; col < x+8; ++col, ++pos) {

int p = row\*width\*comp + col\*comp;

float r, g, b;

if(row >= height) {

p -= width\*comp\*(row+1 - height);

}

if(col >= width) {

p -= comp\*(col+1 - width);

}

r = imageData[p+0];

g = imageData[p+ofsG];

b = imageData[p+ofsB];

YDU[pos]=+0.29900f\*r+0.58700f\*g+0.11400f\*b-128;

UDU[pos]=-0.16874f\*r-0.33126f\*g+0.50000f\*b;

VDU[pos]=+0.50000f\*r-0.41869f\*g-0.08131f\*b;

}

}

DCY = stbiw\_\_jpg\_processDU(s, &bitBuf, &bitCnt, YDU, fdtbl\_Y, DCY, YDC\_HT, YAC\_HT);

DCU = stbiw\_\_jpg\_processDU(s, &bitBuf, &bitCnt, UDU, fdtbl\_UV, DCU, UVDC\_HT, UVAC\_HT);

DCV = stbiw\_\_jpg\_processDU(s, &bitBuf, &bitCnt, VDU, fdtbl\_UV, DCV, UVDC\_HT, UVAC\_HT);

}

}

// Do the bit alignment of the EOI marker

stbiw\_\_jpg\_writeBits(s, &bitBuf, &bitCnt, fillBits);

}

// EOI

stbiw\_\_putc(s, 0xFF);

stbiw\_\_putc(s, 0xD9);

return 1;

}

STBIWDEF int stbi\_write\_jpg\_to\_func(stbi\_write\_func \*func, void \*context, int x, int y, int comp, const void \*data, int quality)

{

stbi\_\_write\_context s;

stbi\_\_start\_write\_callbacks(&s, func, context);

return stbi\_write\_jpg\_core(&s, x, y, comp, (void \*) data, quality);

}

#ifndef STBI\_WRITE\_NO\_STDIO

STBIWDEF int stbi\_write\_jpg(char const \*filename, int x, int y, int comp, const void \*data, int quality)

{

stbi\_\_write\_context s;

if (stbi\_\_start\_write\_file(&s,filename)) {

int r = stbi\_write\_jpg\_core(&s, x, y, comp, data, quality);

stbi\_\_end\_write\_file(&s);

return r;

} else

return 0;

}

#endif

#endif // STB\_IMAGE\_WRITE\_IMPLEMENTATION

/\* Revision history

1.07 (2017-07-24)

doc fix

1.06 (2017-07-23)

writing JPEG (using Jon Olick's code)

1.05 ???

1.04 (2017-03-03)

monochrome BMP expansion

1.03 ???

1.02 (2016-04-02)

avoid allocating large structures on the stack

1.01 (2016-01-16)

STBIW\_REALLOC\_SIZED: support allocators with no realloc support

avoid race-condition in crc initialization

minor compile issues

1.00 (2015-09-14)

installable file IO function

0.99 (2015-09-13)

warning fixes; TGA rle support

0.98 (2015-04-08)

added STBIW\_MALLOC, STBIW\_ASSERT etc

0.97 (2015-01-18)

fixed HDR asserts, rewrote HDR rle logic

0.96 (2015-01-17)

add HDR output

fix monochrome BMP

0.95 (2014-08-17)

add monochrome TGA output

0.94 (2014-05-31)

rename private functions to avoid conflicts with stb\_image.h

0.93 (2014-05-27)

warning fixes

0.92 (2010-08-01)

casts to unsigned char to fix warnings

0.91 (2010-07-17)

first public release

0.90 first internal release

\*/

/\*

------------------------------------------------------------------------------

This software is available under 2 licenses -- choose whichever you prefer.

------------------------------------------------------------------------------

ALTERNATIVE A - MIT License

Copyright (c) 2017 Sean Barrett

Permission is hereby granted, free of charge, to any person obtaining a copy of

this software and associated documentation files (the "Software"), to deal in

the Software without restriction, including without limitation the rights to

use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies

of the Software, and to permit persons to whom the Software is furnished to do

so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

------------------------------------------------------------------------------

ALTERNATIVE B - Public Domain (www.unlicense.org)

This is free and unencumbered software released into the public domain.

Anyone is free to copy, modify, publish, use, compile, sell, or distribute this

software, either in source code form or as a compiled binary, for any purpose,

commercial or non-commercial, and by any means.

In jurisdictions that recognize copyright laws, the author or authors of this

software dedicate any and all copyright interest in the software to the public

domain. We make this dedication for the benefit of the public at large and to

the detriment of our heirs and successors. We intend this dedication to be an

overt act of relinquishment in perpetuity of all present and future rights to

this software under copyright law.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN

ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION

WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

------------------------------------------------------------------------------

\*/