/\* stb\_image - v2.16 - public domain image loader - http://nothings.org/stb\_image.h

no warranty implied; use at your own risk

Do this:

#define STB\_IMAGE\_IMPLEMENTATION

before you include this file in \*one\* C or C++ file to create the implementation.

// i.e. it should look like this:

#include ...

#include ...

#include ...

#define STB\_IMAGE\_IMPLEMENTATION

#include "stb\_image.h"

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

And #define STBI\_MALLOC, STBI\_REALLOC, and STBI\_FREE to avoid using malloc,realloc,free

QUICK NOTES:

Primarily of interest to game developers and other people who can

avoid problematic images and only need the trivial interface

JPEG baseline & progressive (12 bpc/arithmetic not supported, same as stock IJG lib)

PNG 1/2/4/8/16-bit-per-channel

TGA (not sure what subset, if a subset)

BMP non-1bpp, non-RLE

PSD (composited view only, no extra channels, 8/16 bit-per-channel)

GIF (\*comp always reports as 4-channel)

HDR (radiance rgbE format)

PIC (Softimage PIC)

PNM (PPM and PGM binary only)

Animated GIF still needs a proper API, but here's one way to do it:

http://gist.github.com/urraka/685d9a6340b26b830d49

- decode from memory or through FILE (define STBI\_NO\_STDIO to remove code)

- decode from arbitrary I/O callbacks

- SIMD acceleration on x86/x64 (SSE2) and ARM (NEON)

Full documentation under "DOCUMENTATION" below.

LICENSE

See end of file for license information.

RECENT REVISION HISTORY:

2.16 (2017-07-23) all functions have 16-bit variants; optimizations; bugfixes

2.15 (2017-03-18) fix png-1,2,4; all Imagenet JPGs; no runtime SSE detection on GCC

2.14 (2017-03-03) remove deprecated STBI\_JPEG\_OLD; fixes for Imagenet JPGs

2.13 (2016-12-04) experimental 16-bit API, only for PNG so far; fixes

2.12 (2016-04-02) fix typo in 2.11 PSD fix that caused crashes

2.11 (2016-04-02) 16-bit PNGS; enable SSE2 in non-gcc x64

RGB-format JPEG; remove white matting in PSD;

allocate large structures on the stack;

correct channel count for PNG & BMP

2.10 (2016-01-22) avoid warning introduced in 2.09

2.09 (2016-01-16) 16-bit TGA; comments in PNM files; STBI\_REALLOC\_SIZED

See end of file for full revision history.

============================ Contributors =========================

Image formats Extensions, features

Sean Barrett (jpeg, png, bmp) Jetro Lauha (stbi\_info)

Nicolas Schulz (hdr, psd) Martin "SpartanJ" Golini (stbi\_info)

Jonathan Dummer (tga) James "moose2000" Brown (iPhone PNG)

Jean-Marc Lienher (gif) Ben "Disch" Wenger (io callbacks)

Tom Seddon (pic) Omar Cornut (1/2/4-bit PNG)

Thatcher Ulrich (psd) Nicolas Guillemot (vertical flip)

Ken Miller (pgm, ppm) Richard Mitton (16-bit PSD)

github:urraka (animated gif) Junggon Kim (PNM comments)

Daniel Gibson (16-bit TGA)

socks-the-fox (16-bit PNG)

Jeremy Sawicki (handle all ImageNet JPGs)

Optimizations & bugfixes

Fabian "ryg" Giesen

Arseny Kapoulkine

John-Mark Allen

Bug & warning fixes

Marc LeBlanc David Woo Guillaume George Martins Mozeiko

Christpher Lloyd Jerry Jansson Joseph Thomson Phil Jordan

Dave Moore Roy Eltham Hayaki Saito Nathan Reed

Won Chun Luke Graham Johan Duparc Nick Verigakis

the Horde3D community Thomas Ruf Ronny Chevalier Baldur Karlsson

Janez Zemva John Bartholomew Michal Cichon github:rlyeh

Jonathan Blow Ken Hamada Tero Hanninen github:romigrou

Laurent Gomila Cort Stratton Sergio Gonzalez github:svdijk

Aruelien Pocheville Thibault Reuille Cass Everitt github:snagar

Ryamond Barbiero Paul Du Bois Engin Manap github:Zelex

Michaelangel007@github Philipp Wiesemann Dale Weiler github:grim210

Oriol Ferrer Mesia Josh Tobin Matthew Gregan github:sammyhw

Blazej Dariusz Roszkowski Gregory Mullen github:phprus

Christian Floisand Kevin Schmidt github:poppolopoppo

\*/

#ifndef STBI\_INCLUDE\_STB\_IMAGE\_H

#define STBI\_INCLUDE\_STB\_IMAGE\_H

// DOCUMENTATION

//

// Limitations:

// - no 16-bit-per-channel PNG

// - no 12-bit-per-channel JPEG

// - no JPEGs with arithmetic coding

// - no 1-bit BMP

// - GIF always returns \*comp=4

//

// Basic usage (see HDR discussion below for HDR usage):

// int x,y,n;

// unsigned char \*data = stbi\_load(filename, &x, &y, &n, 0);

// // ... process data if not NULL ...

// // ... x = width, y = height, n = # 8-bit components per pixel ...

// // ... replace '0' with '1'..'4' to force that many components per pixel

// // ... but 'n' will always be the number that it would have been if you said 0

// stbi\_image\_free(data)

//

// Standard parameters:

// int \*x -- outputs image width in pixels

// int \*y -- outputs image height in pixels

// int \*channels\_in\_file -- outputs # of image components in image file

// int desired\_channels -- if non-zero, # of image components requested in result

//

// The return value from an image loader is an 'unsigned char \*' which points

// to the pixel data, or NULL on an allocation failure or if the image is

// corrupt or invalid. The pixel data consists of \*y scanlines of \*x pixels,

// with each pixel consisting of N interleaved 8-bit components; the first

// pixel pointed to is top-left-most in the image. There is no padding between

// image scanlines or between pixels, regardless of format. The number of

// components N is 'desired\_channels' if desired\_channels is non-zero, or

// \*channels\_in\_file otherwise. If desired\_channels is non-zero,

// \*channels\_in\_file has the number of components that \_would\_ have been

// output otherwise. E.g. if you set desired\_channels to 4, you will always

// get RGBA output, but you can check \*channels\_in\_file to see if it's trivially

// opaque because e.g. there were only 3 channels in the source image.

//

// An output image with N components has the following components interleaved

// in this order in each pixel:

//

// N=#comp components

// 1 grey

// 2 grey, alpha

// 3 red, green, blue

// 4 red, green, blue, alpha

//

// If image loading fails for any reason, the return value will be NULL,

// and \*x, \*y, \*channels\_in\_file will be unchanged. The function

// stbi\_failure\_reason() can be queried for an extremely brief, end-user

// unfriendly explanation of why the load failed. Define STBI\_NO\_FAILURE\_STRINGS

// to avoid compiling these strings at all, and STBI\_FAILURE\_USERMSG to get slightly

// more user-friendly ones.

//

// Paletted PNG, BMP, GIF, and PIC images are automatically depalettized.

//

// ===========================================================================

//

// Philosophy

//

// stb libraries are designed with the following priorities:

//

// 1. easy to use

// 2. easy to maintain

// 3. good performance

//

// Sometimes I let "good performance" creep up in priority over "easy to maintain",

// and for best performance I may provide less-easy-to-use APIs that give higher

// performance, in addition to the easy to use ones. Nevertheless, it's important

// to keep in mind that from the standpoint of you, a client of this library,

// all you care about is #1 and #3, and stb libraries DO NOT emphasize #3 above all.

//

// Some secondary priorities arise directly from the first two, some of which

// make more explicit reasons why performance can't be emphasized.

//

// - Portable ("ease of use")

// - Small source code footprint ("easy to maintain")

// - No dependencies ("ease of use")

//

// ===========================================================================

//

// I/O callbacks

//

// I/O callbacks allow you to read from arbitrary sources, like packaged

// files or some other source. Data read from callbacks are processed

// through a small internal buffer (currently 128 bytes) to try to reduce

// overhead.

//

// The three functions you must define are "read" (reads some bytes of data),

// "skip" (skips some bytes of data), "eof" (reports if the stream is at the end).

//

// ===========================================================================

//

// SIMD support

//

// The JPEG decoder will try to automatically use SIMD kernels on x86 when

// supported by the compiler. For ARM Neon support, you must explicitly

// request it.

//

// (The old do-it-yourself SIMD API is no longer supported in the current

// code.)

//

// On x86, SSE2 will automatically be used when available based on a run-time

// test; if not, the generic C versions are used as a fall-back. On ARM targets,

// the typical path is to have separate builds for NEON and non-NEON devices

// (at least this is true for iOS and Android). Therefore, the NEON support is

// toggled by a build flag: define STBI\_NEON to get NEON loops.

//

// If for some reason you do not want to use any of SIMD code, or if

// you have issues compiling it, you can disable it entirely by

// defining STBI\_NO\_SIMD.

//

// ===========================================================================

//

// HDR image support (disable by defining STBI\_NO\_HDR)

//

// stb\_image now supports loading HDR images in general, and currently

// the Radiance .HDR file format, although the support is provided

// generically. You can still load any file through the existing interface;

// if you attempt to load an HDR file, it will be automatically remapped to

// LDR, assuming gamma 2.2 and an arbitrary scale factor defaulting to 1;

// both of these constants can be reconfigured through this interface:

//

// stbi\_hdr\_to\_ldr\_gamma(2.2f);

// stbi\_hdr\_to\_ldr\_scale(1.0f);

//

// (note, do not use \_inverse\_ constants; stbi\_image will invert them

// appropriately).

//

// Additionally, there is a new, parallel interface for loading files as

// (linear) floats to preserve the full dynamic range:

//

// float \*data = stbi\_loadf(filename, &x, &y, &n, 0);

//

// If you load LDR images through this interface, those images will

// be promoted to floating point values, run through the inverse of

// constants corresponding to the above:

//

// stbi\_ldr\_to\_hdr\_scale(1.0f);

// stbi\_ldr\_to\_hdr\_gamma(2.2f);

//

// Finally, given a filename (or an open file or memory block--see header

// file for details) containing image data, you can query for the "most

// appropriate" interface to use (that is, whether the image is HDR or

// not), using:

//

// stbi\_is\_hdr(char \*filename);

//

// ===========================================================================

//

// iPhone PNG support:

//

// By default we convert iphone-formatted PNGs back to RGB, even though

// they are internally encoded differently. You can disable this conversion

// by by calling stbi\_convert\_iphone\_png\_to\_rgb(0), in which case

// you will always just get the native iphone "format" through (which

// is BGR stored in RGB).

//

// Call stbi\_set\_unpremultiply\_on\_load(1) as well to force a divide per

// pixel to remove any premultiplied alpha \*only\* if the image file explicitly

// says there's premultiplied data (currently only happens in iPhone images,

// and only if iPhone convert-to-rgb processing is on).

//

// ===========================================================================

//

// ADDITIONAL CONFIGURATION

//

// - You can suppress implementation of any of the decoders to reduce

// your code footprint by #defining one or more of the following

// symbols before creating the implementation.

//

// STBI\_NO\_JPEG

// STBI\_NO\_PNG

// STBI\_NO\_BMP

// STBI\_NO\_PSD

// STBI\_NO\_TGA

// STBI\_NO\_GIF

// STBI\_NO\_HDR

// STBI\_NO\_PIC

// STBI\_NO\_PNM (.ppm and .pgm)

//

// - You can request \*only\* certain decoders and suppress all other ones

// (this will be more forward-compatible, as addition of new decoders

// doesn't require you to disable them explicitly):

//

// STBI\_ONLY\_JPEG

// STBI\_ONLY\_PNG

// STBI\_ONLY\_BMP

// STBI\_ONLY\_PSD

// STBI\_ONLY\_TGA

// STBI\_ONLY\_GIF

// STBI\_ONLY\_HDR

// STBI\_ONLY\_PIC

// STBI\_ONLY\_PNM (.ppm and .pgm)

//

// - If you use STBI\_NO\_PNG (or \_ONLY\_ without PNG), and you still

// want the zlib decoder to be available, #define STBI\_SUPPORT\_ZLIB

//

#ifndef STBI\_NO\_STDIO

#include <stdio.h>

#endif // STBI\_NO\_STDIO

#define STBI\_VERSION 1

enum

{

STBI\_default = 0, // only used for desired\_channels

STBI\_grey = 1,

STBI\_grey\_alpha = 2,

STBI\_rgb = 3,

STBI\_rgb\_alpha = 4

};

typedef unsigned char stbi\_uc;

typedef unsigned short stbi\_us;

#ifdef \_\_cplusplus

extern "C" {

#endif

#ifdef STB\_IMAGE\_STATIC

#define STBIDEF static

#else

#define STBIDEF extern

#endif

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

//

// PRIMARY API - works on images of any type

//

//

// load image by filename, open file, or memory buffer

//

typedef struct

{

int (\*read) (void \*user,char \*data,int size); // fill 'data' with 'size' bytes. return number of bytes actually read

void (\*skip) (void \*user,int n); // skip the next 'n' bytes, or 'unget' the last -n bytes if negative

int (\*eof) (void \*user); // returns nonzero if we are at end of file/data

} stbi\_io\_callbacks;

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

//

// 8-bits-per-channel interface

//

STBIDEF stbi\_uc \*stbi\_load\_from\_memory (stbi\_uc const \*buffer, int len , int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

STBIDEF stbi\_uc \*stbi\_load\_from\_callbacks(stbi\_io\_callbacks const \*clbk , void \*user, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

#ifndef STBI\_NO\_STDIO

STBIDEF stbi\_uc \*stbi\_load (char const \*filename, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

STBIDEF stbi\_uc \*stbi\_load\_from\_file (FILE \*f, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

// for stbi\_load\_from\_file, file pointer is left pointing immediately after image

#endif

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

//

// 16-bits-per-channel interface

//

STBIDEF stbi\_us \*stbi\_load\_16\_from\_memory (stbi\_uc const \*buffer, int len, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

STBIDEF stbi\_us \*stbi\_load\_16\_from\_callbacks(stbi\_io\_callbacks const \*clbk, void \*user, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

#ifndef STBI\_NO\_STDIO

STBIDEF stbi\_us \*stbi\_load\_16 (char const \*filename, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

STBIDEF stbi\_us \*stbi\_load\_from\_file\_16(FILE \*f, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

#endif

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

//

// float-per-channel interface

//

#ifndef STBI\_NO\_LINEAR

STBIDEF float \*stbi\_loadf\_from\_memory (stbi\_uc const \*buffer, int len, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

STBIDEF float \*stbi\_loadf\_from\_callbacks (stbi\_io\_callbacks const \*clbk, void \*user, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

#ifndef STBI\_NO\_STDIO

STBIDEF float \*stbi\_loadf (char const \*filename, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

STBIDEF float \*stbi\_loadf\_from\_file (FILE \*f, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels);

#endif

#endif

#ifndef STBI\_NO\_HDR

STBIDEF void stbi\_hdr\_to\_ldr\_gamma(float gamma);

STBIDEF void stbi\_hdr\_to\_ldr\_scale(float scale);

#endif // STBI\_NO\_HDR

#ifndef STBI\_NO\_LINEAR

STBIDEF void stbi\_ldr\_to\_hdr\_gamma(float gamma);

STBIDEF void stbi\_ldr\_to\_hdr\_scale(float scale);

#endif // STBI\_NO\_LINEAR

// stbi\_is\_hdr is always defined, but always returns false if STBI\_NO\_HDR

STBIDEF int stbi\_is\_hdr\_from\_callbacks(stbi\_io\_callbacks const \*clbk, void \*user);

STBIDEF int stbi\_is\_hdr\_from\_memory(stbi\_uc const \*buffer, int len);

#ifndef STBI\_NO\_STDIO

STBIDEF int stbi\_is\_hdr (char const \*filename);

STBIDEF int stbi\_is\_hdr\_from\_file(FILE \*f);

#endif // STBI\_NO\_STDIO

// get a VERY brief reason for failure

// NOT THREADSAFE

STBIDEF const char \*stbi\_failure\_reason (void);

// free the loaded image -- this is just free()

STBIDEF void stbi\_image\_free (void \*retval\_from\_stbi\_load);

// get image dimensions & components without fully decoding

STBIDEF int stbi\_info\_from\_memory(stbi\_uc const \*buffer, int len, int \*x, int \*y, int \*comp);

STBIDEF int stbi\_info\_from\_callbacks(stbi\_io\_callbacks const \*clbk, void \*user, int \*x, int \*y, int \*comp);

#ifndef STBI\_NO\_STDIO

STBIDEF int stbi\_info (char const \*filename, int \*x, int \*y, int \*comp);

STBIDEF int stbi\_info\_from\_file (FILE \*f, int \*x, int \*y, int \*comp);

#endif

// for image formats that explicitly notate that they have premultiplied alpha,

// we just return the colors as stored in the file. set this flag to force

// unpremultiplication. results are undefined if the unpremultiply overflow.

STBIDEF void stbi\_set\_unpremultiply\_on\_load(int flag\_true\_if\_should\_unpremultiply);

// indicate whether we should process iphone images back to canonical format,

// or just pass them through "as-is"

STBIDEF void stbi\_convert\_iphone\_png\_to\_rgb(int flag\_true\_if\_should\_convert);

// flip the image vertically, so the first pixel in the output array is the bottom left

STBIDEF void stbi\_set\_flip\_vertically\_on\_load(int flag\_true\_if\_should\_flip);

// ZLIB client - used by PNG, available for other purposes

STBIDEF char \*stbi\_zlib\_decode\_malloc\_guesssize(const char \*buffer, int len, int initial\_size, int \*outlen);

STBIDEF char \*stbi\_zlib\_decode\_malloc\_guesssize\_headerflag(const char \*buffer, int len, int initial\_size, int \*outlen, int parse\_header);

STBIDEF char \*stbi\_zlib\_decode\_malloc(const char \*buffer, int len, int \*outlen);

STBIDEF int stbi\_zlib\_decode\_buffer(char \*obuffer, int olen, const char \*ibuffer, int ilen);

STBIDEF char \*stbi\_zlib\_decode\_noheader\_malloc(const char \*buffer, int len, int \*outlen);

STBIDEF int stbi\_zlib\_decode\_noheader\_buffer(char \*obuffer, int olen, const char \*ibuffer, int ilen);

#ifdef \_\_cplusplus

}

#endif

//

//

//// end header file /////////////////////////////////////////////////////

#endif // STBI\_INCLUDE\_STB\_IMAGE\_H

#ifdef STB\_IMAGE\_IMPLEMENTATION

#if defined(STBI\_ONLY\_JPEG) || defined(STBI\_ONLY\_PNG) || defined(STBI\_ONLY\_BMP) \

|| defined(STBI\_ONLY\_TGA) || defined(STBI\_ONLY\_GIF) || defined(STBI\_ONLY\_PSD) \

|| defined(STBI\_ONLY\_HDR) || defined(STBI\_ONLY\_PIC) || defined(STBI\_ONLY\_PNM) \

|| defined(STBI\_ONLY\_ZLIB)

#ifndef STBI\_ONLY\_JPEG

#define STBI\_NO\_JPEG

#endif

#ifndef STBI\_ONLY\_PNG

#define STBI\_NO\_PNG

#endif

#ifndef STBI\_ONLY\_BMP

#define STBI\_NO\_BMP

#endif

#ifndef STBI\_ONLY\_PSD

#define STBI\_NO\_PSD

#endif

#ifndef STBI\_ONLY\_TGA

#define STBI\_NO\_TGA

#endif

#ifndef STBI\_ONLY\_GIF

#define STBI\_NO\_GIF

#endif

#ifndef STBI\_ONLY\_HDR

#define STBI\_NO\_HDR

#endif

#ifndef STBI\_ONLY\_PIC

#define STBI\_NO\_PIC

#endif

#ifndef STBI\_ONLY\_PNM

#define STBI\_NO\_PNM

#endif

#endif

#if defined(STBI\_NO\_PNG) && !defined(STBI\_SUPPORT\_ZLIB) && !defined(STBI\_NO\_ZLIB)

#define STBI\_NO\_ZLIB

#endif

#include <stdarg.h>

#include <stddef.h> // ptrdiff\_t on osx

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#if !defined(STBI\_NO\_LINEAR) || !defined(STBI\_NO\_HDR)

#include <math.h> // ldexp

#endif

#ifndef STBI\_NO\_STDIO

#include <stdio.h>

#endif

#ifndef STBI\_ASSERT

#include <assert.h>

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

#endif

#ifndef \_MSC\_VER

#ifdef \_\_cplusplus

#define stbi\_inline inline

#else

#define stbi\_inline

#endif

#else

#define stbi\_inline \_\_forceinline

#endif

#ifdef \_MSC\_VER

typedef unsigned short stbi\_\_uint16;

typedef signed short stbi\_\_int16;

typedef unsigned int stbi\_\_uint32;

typedef signed int stbi\_\_int32;

#else

#include <stdint.h>

typedef uint16\_t stbi\_\_uint16;

typedef int16\_t stbi\_\_int16;

typedef uint32\_t stbi\_\_uint32;

typedef int32\_t stbi\_\_int32;

#endif

// should produce compiler error if size is wrong

typedef unsigned char validate\_uint32[sizeof(stbi\_\_uint32)==4 ? 1 : -1];

#ifdef \_MSC\_VER

#define STBI\_NOTUSED(v) (void)(v)

#else

#define STBI\_NOTUSED(v) (void)sizeof(v)

#endif

#ifdef \_MSC\_VER

#define STBI\_HAS\_LROTL

#endif

#ifdef STBI\_HAS\_LROTL

#define stbi\_lrot(x,y) \_lrotl(x,y)

#else

#define stbi\_lrot(x,y) (((x) << (y)) | ((x) >> (32 - (y))))

#endif

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

// ok

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

// ok

#else

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

#endif

#ifndef STBI\_MALLOC

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

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

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

#endif

#ifndef STBI\_REALLOC\_SIZED

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

#endif

// x86/x64 detection

#if defined(\_\_x86\_64\_\_) || defined(\_M\_X64)

#define STBI\_\_X64\_TARGET

#elif defined(\_\_i386) || defined(\_M\_IX86)

#define STBI\_\_X86\_TARGET

#endif

#if defined(\_\_GNUC\_\_) && defined(STBI\_\_X86\_TARGET) && !defined(\_\_SSE2\_\_) && !defined(STBI\_NO\_SIMD)

// gcc doesn't support sse2 intrinsics unless you compile with -msse2,

// which in turn means it gets to use SSE2 everywhere. This is unfortunate,

// but previous attempts to provide the SSE2 functions with runtime

// detection caused numerous issues. The way architecture extensions are

// exposed in GCC/Clang is, sadly, not really suited for one-file libs.

// New behavior: if compiled with -msse2, we use SSE2 without any

// detection; if not, we don't use it at all.

#define STBI\_NO\_SIMD

#endif

#if defined(\_\_MINGW32\_\_) && defined(STBI\_\_X86\_TARGET) && !defined(STBI\_MINGW\_ENABLE\_SSE2) && !defined(STBI\_NO\_SIMD)

// Note that \_\_MINGW32\_\_ doesn't actually mean 32-bit, so we have to avoid STBI\_\_X64\_TARGET

//

// 32-bit MinGW wants ESP to be 16-byte aligned, but this is not in the

// Windows ABI and VC++ as well as Windows DLLs don't maintain that invariant.

// As a result, enabling SSE2 on 32-bit MinGW is dangerous when not

// simultaneously enabling "-mstackrealign".

//

// See https://github.com/nothings/stb/issues/81 for more information.

//

// So default to no SSE2 on 32-bit MinGW. If you've read this far and added

// -mstackrealign to your build settings, feel free to #define STBI\_MINGW\_ENABLE\_SSE2.

#define STBI\_NO\_SIMD

#endif

#if !defined(STBI\_NO\_SIMD) && (defined(STBI\_\_X86\_TARGET) || defined(STBI\_\_X64\_TARGET))

#define STBI\_SSE2

#include <emmintrin.h>

#ifdef \_MSC\_VER

#if \_MSC\_VER >= 1400 // not VC6

#include <intrin.h> // \_\_cpuid

static int stbi\_\_cpuid3(void)

{

int info[4];

\_\_cpuid(info,1);

return info[3];

}

#else

static int stbi\_\_cpuid3(void)

{

int res;

\_\_asm {

mov eax,1

cpuid

mov res,edx

}

return res;

}

#endif

#define STBI\_SIMD\_ALIGN(type, name) \_\_declspec(align(16)) type name

static int stbi\_\_sse2\_available(void)

{

int info3 = stbi\_\_cpuid3();

return ((info3 >> 26) & 1) != 0;

}

#else // assume GCC-style if not VC++

#define STBI\_SIMD\_ALIGN(type, name) type name \_\_attribute\_\_((aligned(16)))

static int stbi\_\_sse2\_available(void)

{

// If we're even attempting to compile this on GCC/Clang, that means

// -msse2 is on, which means the compiler is allowed to use SSE2

// instructions at will, and so are we.

return 1;

}

#endif

#endif

// ARM NEON

#if defined(STBI\_NO\_SIMD) && defined(STBI\_NEON)

#undef STBI\_NEON

#endif

#ifdef STBI\_NEON

#include <arm\_neon.h>

// assume GCC or Clang on ARM targets

#define STBI\_SIMD\_ALIGN(type, name) type name \_\_attribute\_\_((aligned(16)))

#endif

#ifndef STBI\_SIMD\_ALIGN

#define STBI\_SIMD\_ALIGN(type, name) type name

#endif

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

//

// stbi\_\_context struct and start\_xxx functions

// stbi\_\_context structure is our basic context used by all images, so it

// contains all the IO context, plus some basic image information

typedef struct

{

stbi\_\_uint32 img\_x, img\_y;

int img\_n, img\_out\_n;

stbi\_io\_callbacks io;

void \*io\_user\_data;

int read\_from\_callbacks;

int buflen;

stbi\_uc buffer\_start[128];

stbi\_uc \*img\_buffer, \*img\_buffer\_end;

stbi\_uc \*img\_buffer\_original, \*img\_buffer\_original\_end;

} stbi\_\_context;

static void stbi\_\_refill\_buffer(stbi\_\_context \*s);

// initialize a memory-decode context

static void stbi\_\_start\_mem(stbi\_\_context \*s, stbi\_uc const \*buffer, int len)

{

s->io.read = NULL;

s->read\_from\_callbacks = 0;

s->img\_buffer = s->img\_buffer\_original = (stbi\_uc \*) buffer;

s->img\_buffer\_end = s->img\_buffer\_original\_end = (stbi\_uc \*) buffer+len;

}

// initialize a callback-based context

static void stbi\_\_start\_callbacks(stbi\_\_context \*s, stbi\_io\_callbacks \*c, void \*user)

{

s->io = \*c;

s->io\_user\_data = user;

s->buflen = sizeof(s->buffer\_start);

s->read\_from\_callbacks = 1;

s->img\_buffer\_original = s->buffer\_start;

stbi\_\_refill\_buffer(s);

s->img\_buffer\_original\_end = s->img\_buffer\_end;

}

#ifndef STBI\_NO\_STDIO

static int stbi\_\_stdio\_read(void \*user, char \*data, int size)

{

return (int) fread(data,1,size,(FILE\*) user);

}

static void stbi\_\_stdio\_skip(void \*user, int n)

{

fseek((FILE\*) user, n, SEEK\_CUR);

}

static int stbi\_\_stdio\_eof(void \*user)

{

return feof((FILE\*) user);

}

static stbi\_io\_callbacks stbi\_\_stdio\_callbacks =

{

stbi\_\_stdio\_read,

stbi\_\_stdio\_skip,

stbi\_\_stdio\_eof,

};

static void stbi\_\_start\_file(stbi\_\_context \*s, FILE \*f)

{

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

}

//static void stop\_file(stbi\_\_context \*s) { }

#endif // !STBI\_NO\_STDIO

static void stbi\_\_rewind(stbi\_\_context \*s)

{

// conceptually rewind SHOULD rewind to the beginning of the stream,

// but we just rewind to the beginning of the initial buffer, because

// we only use it after doing 'test', which only ever looks at at most 92 bytes

s->img\_buffer = s->img\_buffer\_original;

s->img\_buffer\_end = s->img\_buffer\_original\_end;

}

enum

{

STBI\_ORDER\_RGB,

STBI\_ORDER\_BGR

};

typedef struct

{

int bits\_per\_channel;

int num\_channels;

int channel\_order;

} stbi\_\_result\_info;

#ifndef STBI\_NO\_JPEG

static int stbi\_\_jpeg\_test(stbi\_\_context \*s);

static void \*stbi\_\_jpeg\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri);

static int stbi\_\_jpeg\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp);

#endif

#ifndef STBI\_NO\_PNG

static int stbi\_\_png\_test(stbi\_\_context \*s);

static void \*stbi\_\_png\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri);

static int stbi\_\_png\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp);

#endif

#ifndef STBI\_NO\_BMP

static int stbi\_\_bmp\_test(stbi\_\_context \*s);

static void \*stbi\_\_bmp\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri);

static int stbi\_\_bmp\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp);

#endif

#ifndef STBI\_NO\_TGA

static int stbi\_\_tga\_test(stbi\_\_context \*s);

static void \*stbi\_\_tga\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri);

static int stbi\_\_tga\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp);

#endif

#ifndef STBI\_NO\_PSD

static int stbi\_\_psd\_test(stbi\_\_context \*s);

static void \*stbi\_\_psd\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri, int bpc);

static int stbi\_\_psd\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp);

#endif

#ifndef STBI\_NO\_HDR

static int stbi\_\_hdr\_test(stbi\_\_context \*s);

static float \*stbi\_\_hdr\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri);

static int stbi\_\_hdr\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp);

#endif

#ifndef STBI\_NO\_PIC

static int stbi\_\_pic\_test(stbi\_\_context \*s);

static void \*stbi\_\_pic\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri);

static int stbi\_\_pic\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp);

#endif

#ifndef STBI\_NO\_GIF

static int stbi\_\_gif\_test(stbi\_\_context \*s);

static void \*stbi\_\_gif\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri);

static int stbi\_\_gif\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp);

#endif

#ifndef STBI\_NO\_PNM

static int stbi\_\_pnm\_test(stbi\_\_context \*s);

static void \*stbi\_\_pnm\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri);

static int stbi\_\_pnm\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp);

#endif

// this is not threadsafe

static const char \*stbi\_\_g\_failure\_reason;

STBIDEF const char \*stbi\_failure\_reason(void)

{

return stbi\_\_g\_failure\_reason;

}

static int stbi\_\_err(const char \*str)

{

stbi\_\_g\_failure\_reason = str;

return 0;

}

static void \*stbi\_\_malloc(size\_t size)

{

return STBI\_MALLOC(size);

}

// stb\_image uses ints pervasively, including for offset calculations.

// therefore the largest decoded image size we can support with the

// current code, even on 64-bit targets, is INT\_MAX. this is not a

// significant limitation for the intended use case.

//

// we do, however, need to make sure our size calculations don't

// overflow. hence a few helper functions for size calculations that

// multiply integers together, making sure that they're non-negative

// and no overflow occurs.

// return 1 if the sum is valid, 0 on overflow.

// negative terms are considered invalid.

static int stbi\_\_addsizes\_valid(int a, int b)

{

if (b < 0) return 0;

// now 0 <= b <= INT\_MAX, hence also

// 0 <= INT\_MAX - b <= INTMAX.

// And "a + b <= INT\_MAX" (which might overflow) is the

// same as a <= INT\_MAX - b (no overflow)

return a <= INT\_MAX - b;

}

// returns 1 if the product is valid, 0 on overflow.

// negative factors are considered invalid.

static int stbi\_\_mul2sizes\_valid(int a, int b)

{

if (a < 0 || b < 0) return 0;

if (b == 0) return 1; // mul-by-0 is always safe

// portable way to check for no overflows in a\*b

return a <= INT\_MAX/b;

}

// returns 1 if "a\*b + add" has no negative terms/factors and doesn't overflow

static int stbi\_\_mad2sizes\_valid(int a, int b, int add)

{

return stbi\_\_mul2sizes\_valid(a, b) && stbi\_\_addsizes\_valid(a\*b, add);

}

// returns 1 if "a\*b\*c + add" has no negative terms/factors and doesn't overflow

static int stbi\_\_mad3sizes\_valid(int a, int b, int c, int add)

{

return stbi\_\_mul2sizes\_valid(a, b) && stbi\_\_mul2sizes\_valid(a\*b, c) &&

stbi\_\_addsizes\_valid(a\*b\*c, add);

}

// returns 1 if "a\*b\*c\*d + add" has no negative terms/factors and doesn't overflow

static int stbi\_\_mad4sizes\_valid(int a, int b, int c, int d, int add)

{

return stbi\_\_mul2sizes\_valid(a, b) && stbi\_\_mul2sizes\_valid(a\*b, c) &&

stbi\_\_mul2sizes\_valid(a\*b\*c, d) && stbi\_\_addsizes\_valid(a\*b\*c\*d, add);

}

// mallocs with size overflow checking

static void \*stbi\_\_malloc\_mad2(int a, int b, int add)

{

if (!stbi\_\_mad2sizes\_valid(a, b, add)) return NULL;

return stbi\_\_malloc(a\*b + add);

}

static void \*stbi\_\_malloc\_mad3(int a, int b, int c, int add)

{

if (!stbi\_\_mad3sizes\_valid(a, b, c, add)) return NULL;

return stbi\_\_malloc(a\*b\*c + add);

}

static void \*stbi\_\_malloc\_mad4(int a, int b, int c, int d, int add)

{

if (!stbi\_\_mad4sizes\_valid(a, b, c, d, add)) return NULL;

return stbi\_\_malloc(a\*b\*c\*d + add);

}

// stbi\_\_err - error

// stbi\_\_errpf - error returning pointer to float

// stbi\_\_errpuc - error returning pointer to unsigned char

#ifdef STBI\_NO\_FAILURE\_STRINGS

#define stbi\_\_err(x,y) 0

#elif defined(STBI\_FAILURE\_USERMSG)

#define stbi\_\_err(x,y) stbi\_\_err(y)

#else

#define stbi\_\_err(x,y) stbi\_\_err(x)

#endif

#define stbi\_\_errpf(x,y) ((float \*)(size\_t) (stbi\_\_err(x,y)?NULL:NULL))

#define stbi\_\_errpuc(x,y) ((unsigned char \*)(size\_t) (stbi\_\_err(x,y)?NULL:NULL))

STBIDEF void stbi\_image\_free(void \*retval\_from\_stbi\_load)

{

STBI\_FREE(retval\_from\_stbi\_load);

}

#ifndef STBI\_NO\_LINEAR

static float \*stbi\_\_ldr\_to\_hdr(stbi\_uc \*data, int x, int y, int comp);

#endif

#ifndef STBI\_NO\_HDR

static stbi\_uc \*stbi\_\_hdr\_to\_ldr(float \*data, int x, int y, int comp);

#endif

static int stbi\_\_vertically\_flip\_on\_load = 0;

STBIDEF void stbi\_set\_flip\_vertically\_on\_load(int flag\_true\_if\_should\_flip)

{

stbi\_\_vertically\_flip\_on\_load = flag\_true\_if\_should\_flip;

}

static void \*stbi\_\_load\_main(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri, int bpc)

{

memset(ri, 0, sizeof(\*ri)); // make sure it's initialized if we add new fields

ri->bits\_per\_channel = 8; // default is 8 so most paths don't have to be changed

ri->channel\_order = STBI\_ORDER\_RGB; // all current input & output are this, but this is here so we can add BGR order

ri->num\_channels = 0;

#ifndef STBI\_NO\_JPEG

if (stbi\_\_jpeg\_test(s)) return stbi\_\_jpeg\_load(s,x,y,comp,req\_comp, ri);

#endif

#ifndef STBI\_NO\_PNG

if (stbi\_\_png\_test(s)) return stbi\_\_png\_load(s,x,y,comp,req\_comp, ri);

#endif

#ifndef STBI\_NO\_BMP

if (stbi\_\_bmp\_test(s)) return stbi\_\_bmp\_load(s,x,y,comp,req\_comp, ri);

#endif

#ifndef STBI\_NO\_GIF

if (stbi\_\_gif\_test(s)) return stbi\_\_gif\_load(s,x,y,comp,req\_comp, ri);

#endif

#ifndef STBI\_NO\_PSD

if (stbi\_\_psd\_test(s)) return stbi\_\_psd\_load(s,x,y,comp,req\_comp, ri, bpc);

#endif

#ifndef STBI\_NO\_PIC

if (stbi\_\_pic\_test(s)) return stbi\_\_pic\_load(s,x,y,comp,req\_comp, ri);

#endif

#ifndef STBI\_NO\_PNM

if (stbi\_\_pnm\_test(s)) return stbi\_\_pnm\_load(s,x,y,comp,req\_comp, ri);

#endif

#ifndef STBI\_NO\_HDR

if (stbi\_\_hdr\_test(s)) {

float \*hdr = stbi\_\_hdr\_load(s, x,y,comp,req\_comp, ri);

return stbi\_\_hdr\_to\_ldr(hdr, \*x, \*y, req\_comp ? req\_comp : \*comp);

}

#endif

#ifndef STBI\_NO\_TGA

// test tga last because it's a crappy test!

if (stbi\_\_tga\_test(s))

return stbi\_\_tga\_load(s,x,y,comp,req\_comp, ri);

#endif

return stbi\_\_errpuc("unknown image type", "Image not of any known type, or corrupt");

}

static stbi\_uc \*stbi\_\_convert\_16\_to\_8(stbi\_\_uint16 \*orig, int w, int h, int channels)

{

int i;

int img\_len = w \* h \* channels;

stbi\_uc \*reduced;

reduced = (stbi\_uc \*) stbi\_\_malloc(img\_len);

if (reduced == NULL) return stbi\_\_errpuc("outofmem", "Out of memory");

for (i = 0; i < img\_len; ++i)

reduced[i] = (stbi\_uc)((orig[i] >> 8) & 0xFF); // top half of each byte is sufficient approx of 16->8 bit scaling

STBI\_FREE(orig);

return reduced;

}

static stbi\_\_uint16 \*stbi\_\_convert\_8\_to\_16(stbi\_uc \*orig, int w, int h, int channels)

{

int i;

int img\_len = w \* h \* channels;

stbi\_\_uint16 \*enlarged;

enlarged = (stbi\_\_uint16 \*) stbi\_\_malloc(img\_len\*2);

if (enlarged == NULL) return (stbi\_\_uint16 \*) stbi\_\_errpuc("outofmem", "Out of memory");

for (i = 0; i < img\_len; ++i)

enlarged[i] = (stbi\_\_uint16)((orig[i] << 8) + orig[i]); // replicate to high and low byte, maps 0->0, 255->0xffff

STBI\_FREE(orig);

return enlarged;

}

static void stbi\_\_vertical\_flip(void \*image, int w, int h, int bytes\_per\_pixel)

{

int row;

size\_t bytes\_per\_row = (size\_t)w \* bytes\_per\_pixel;

stbi\_uc temp[2048];

stbi\_uc \*bytes = (stbi\_uc \*)image;

for (row = 0; row < (h>>1); row++) {

stbi\_uc \*row0 = bytes + row\*bytes\_per\_row;

stbi\_uc \*row1 = bytes + (h - row - 1)\*bytes\_per\_row;

// swap row0 with row1

size\_t bytes\_left = bytes\_per\_row;

while (bytes\_left) {

size\_t bytes\_copy = (bytes\_left < sizeof(temp)) ? bytes\_left : sizeof(temp);

memcpy(temp, row0, bytes\_copy);

memcpy(row0, row1, bytes\_copy);

memcpy(row1, temp, bytes\_copy);

row0 += bytes\_copy;

row1 += bytes\_copy;

bytes\_left -= bytes\_copy;

}

}

}

static unsigned char \*stbi\_\_load\_and\_postprocess\_8bit(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp)

{

stbi\_\_result\_info ri;

void \*result = stbi\_\_load\_main(s, x, y, comp, req\_comp, &ri, 8);

if (result == NULL)

return NULL;

if (ri.bits\_per\_channel != 8) {

STBI\_ASSERT(ri.bits\_per\_channel == 16);

result = stbi\_\_convert\_16\_to\_8((stbi\_\_uint16 \*) result, \*x, \*y, req\_comp == 0 ? \*comp : req\_comp);

ri.bits\_per\_channel = 8;

}

// @TODO: move stbi\_\_convert\_format to here

if (stbi\_\_vertically\_flip\_on\_load) {

int channels = req\_comp ? req\_comp : \*comp;

stbi\_\_vertical\_flip(result, \*x, \*y, channels \* sizeof(stbi\_uc));

}

return (unsigned char \*) result;

}

static stbi\_\_uint16 \*stbi\_\_load\_and\_postprocess\_16bit(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp)

{

stbi\_\_result\_info ri;

void \*result = stbi\_\_load\_main(s, x, y, comp, req\_comp, &ri, 16);

if (result == NULL)

return NULL;

if (ri.bits\_per\_channel != 16) {

STBI\_ASSERT(ri.bits\_per\_channel == 8);

result = stbi\_\_convert\_8\_to\_16((stbi\_uc \*) result, \*x, \*y, req\_comp == 0 ? \*comp : req\_comp);

ri.bits\_per\_channel = 16;

}

// @TODO: move stbi\_\_convert\_format16 to here

// @TODO: special case RGB-to-Y (and RGBA-to-YA) for 8-bit-to-16-bit case to keep more precision

if (stbi\_\_vertically\_flip\_on\_load) {

int channels = req\_comp ? req\_comp : \*comp;

stbi\_\_vertical\_flip(result, \*x, \*y, channels \* sizeof(stbi\_\_uint16));

}

return (stbi\_\_uint16 \*) result;

}

#ifndef STBI\_NO\_HDR

static void stbi\_\_float\_postprocess(float \*result, int \*x, int \*y, int \*comp, int req\_comp)

{

if (stbi\_\_vertically\_flip\_on\_load && result != NULL) {

int channels = req\_comp ? req\_comp : \*comp;

stbi\_\_vertical\_flip(result, \*x, \*y, channels \* sizeof(float));

}

}

#endif

#ifndef STBI\_NO\_STDIO

static FILE \*stbi\_\_fopen(char const \*filename, char const \*mode)

{

FILE \*f;

#if defined(\_MSC\_VER) && \_MSC\_VER >= 1400

if (0 != fopen\_s(&f, filename, mode))

f=0;

#else

f = fopen(filename, mode);

#endif

return f;

}

STBIDEF stbi\_uc \*stbi\_load(char const \*filename, int \*x, int \*y, int \*comp, int req\_comp)

{

FILE \*f = stbi\_\_fopen(filename, "rb");

unsigned char \*result;

if (!f) return stbi\_\_errpuc("can't fopen", "Unable to open file");

result = stbi\_load\_from\_file(f,x,y,comp,req\_comp);

fclose(f);

return result;

}

STBIDEF stbi\_uc \*stbi\_load\_from\_file(FILE \*f, int \*x, int \*y, int \*comp, int req\_comp)

{

unsigned char \*result;

stbi\_\_context s;

stbi\_\_start\_file(&s,f);

result = stbi\_\_load\_and\_postprocess\_8bit(&s,x,y,comp,req\_comp);

if (result) {

// need to 'unget' all the characters in the IO buffer

fseek(f, - (int) (s.img\_buffer\_end - s.img\_buffer), SEEK\_CUR);

}

return result;

}

STBIDEF stbi\_\_uint16 \*stbi\_load\_from\_file\_16(FILE \*f, int \*x, int \*y, int \*comp, int req\_comp)

{

stbi\_\_uint16 \*result;

stbi\_\_context s;

stbi\_\_start\_file(&s,f);

result = stbi\_\_load\_and\_postprocess\_16bit(&s,x,y,comp,req\_comp);

if (result) {

// need to 'unget' all the characters in the IO buffer

fseek(f, - (int) (s.img\_buffer\_end - s.img\_buffer), SEEK\_CUR);

}

return result;

}

STBIDEF stbi\_us \*stbi\_load\_16(char const \*filename, int \*x, int \*y, int \*comp, int req\_comp)

{

FILE \*f = stbi\_\_fopen(filename, "rb");

stbi\_\_uint16 \*result;

if (!f) return (stbi\_us \*) stbi\_\_errpuc("can't fopen", "Unable to open file");

result = stbi\_load\_from\_file\_16(f,x,y,comp,req\_comp);

fclose(f);

return result;

}

#endif //!STBI\_NO\_STDIO

STBIDEF stbi\_us \*stbi\_load\_16\_from\_memory(stbi\_uc const \*buffer, int len, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels)

{

stbi\_\_context s;

stbi\_\_start\_mem(&s,buffer,len);

return stbi\_\_load\_and\_postprocess\_16bit(&s,x,y,channels\_in\_file,desired\_channels);

}

STBIDEF stbi\_us \*stbi\_load\_16\_from\_callbacks(stbi\_io\_callbacks const \*clbk, void \*user, int \*x, int \*y, int \*channels\_in\_file, int desired\_channels)

{

stbi\_\_context s;

stbi\_\_start\_callbacks(&s, (stbi\_io\_callbacks \*)clbk, user);

return stbi\_\_load\_and\_postprocess\_16bit(&s,x,y,channels\_in\_file,desired\_channels);

}

STBIDEF stbi\_uc \*stbi\_load\_from\_memory(stbi\_uc const \*buffer, int len, int \*x, int \*y, int \*comp, int req\_comp)

{

stbi\_\_context s;

stbi\_\_start\_mem(&s,buffer,len);

return stbi\_\_load\_and\_postprocess\_8bit(&s,x,y,comp,req\_comp);

}

STBIDEF stbi\_uc \*stbi\_load\_from\_callbacks(stbi\_io\_callbacks const \*clbk, void \*user, int \*x, int \*y, int \*comp, int req\_comp)

{

stbi\_\_context s;

stbi\_\_start\_callbacks(&s, (stbi\_io\_callbacks \*) clbk, user);

return stbi\_\_load\_and\_postprocess\_8bit(&s,x,y,comp,req\_comp);

}

#ifndef STBI\_NO\_LINEAR

static float \*stbi\_\_loadf\_main(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp)

{

unsigned char \*data;

#ifndef STBI\_NO\_HDR

if (stbi\_\_hdr\_test(s)) {

stbi\_\_result\_info ri;

float \*hdr\_data = stbi\_\_hdr\_load(s,x,y,comp,req\_comp, &ri);

if (hdr\_data)

stbi\_\_float\_postprocess(hdr\_data,x,y,comp,req\_comp);

return hdr\_data;

}

#endif

data = stbi\_\_load\_and\_postprocess\_8bit(s, x, y, comp, req\_comp);

if (data)

return stbi\_\_ldr\_to\_hdr(data, \*x, \*y, req\_comp ? req\_comp : \*comp);

return stbi\_\_errpf("unknown image type", "Image not of any known type, or corrupt");

}

STBIDEF float \*stbi\_loadf\_from\_memory(stbi\_uc const \*buffer, int len, int \*x, int \*y, int \*comp, int req\_comp)

{

stbi\_\_context s;

stbi\_\_start\_mem(&s,buffer,len);

return stbi\_\_loadf\_main(&s,x,y,comp,req\_comp);

}

STBIDEF float \*stbi\_loadf\_from\_callbacks(stbi\_io\_callbacks const \*clbk, void \*user, int \*x, int \*y, int \*comp, int req\_comp)

{

stbi\_\_context s;

stbi\_\_start\_callbacks(&s, (stbi\_io\_callbacks \*) clbk, user);

return stbi\_\_loadf\_main(&s,x,y,comp,req\_comp);

}

#ifndef STBI\_NO\_STDIO

STBIDEF float \*stbi\_loadf(char const \*filename, int \*x, int \*y, int \*comp, int req\_comp)

{

float \*result;

FILE \*f = stbi\_\_fopen(filename, "rb");

if (!f) return stbi\_\_errpf("can't fopen", "Unable to open file");

result = stbi\_loadf\_from\_file(f,x,y,comp,req\_comp);

fclose(f);

return result;

}

STBIDEF float \*stbi\_loadf\_from\_file(FILE \*f, int \*x, int \*y, int \*comp, int req\_comp)

{

stbi\_\_context s;

stbi\_\_start\_file(&s,f);

return stbi\_\_loadf\_main(&s,x,y,comp,req\_comp);

}

#endif // !STBI\_NO\_STDIO

#endif // !STBI\_NO\_LINEAR

// these is-hdr-or-not is defined independent of whether STBI\_NO\_LINEAR is

// defined, for API simplicity; if STBI\_NO\_LINEAR is defined, it always

// reports false!

STBIDEF int stbi\_is\_hdr\_from\_memory(stbi\_uc const \*buffer, int len)

{

#ifndef STBI\_NO\_HDR

stbi\_\_context s;

stbi\_\_start\_mem(&s,buffer,len);

return stbi\_\_hdr\_test(&s);

#else

STBI\_NOTUSED(buffer);

STBI\_NOTUSED(len);

return 0;

#endif

}

#ifndef STBI\_NO\_STDIO

STBIDEF int stbi\_is\_hdr (char const \*filename)

{

FILE \*f = stbi\_\_fopen(filename, "rb");

int result=0;

if (f) {

result = stbi\_is\_hdr\_from\_file(f);

fclose(f);

}

return result;

}

STBIDEF int stbi\_is\_hdr\_from\_file(FILE \*f)

{

#ifndef STBI\_NO\_HDR

stbi\_\_context s;

stbi\_\_start\_file(&s,f);

return stbi\_\_hdr\_test(&s);

#else

STBI\_NOTUSED(f);

return 0;

#endif

}

#endif // !STBI\_NO\_STDIO

STBIDEF int stbi\_is\_hdr\_from\_callbacks(stbi\_io\_callbacks const \*clbk, void \*user)

{

#ifndef STBI\_NO\_HDR

stbi\_\_context s;

stbi\_\_start\_callbacks(&s, (stbi\_io\_callbacks \*) clbk, user);

return stbi\_\_hdr\_test(&s);

#else

STBI\_NOTUSED(clbk);

STBI\_NOTUSED(user);

return 0;

#endif

}

#ifndef STBI\_NO\_LINEAR

static float stbi\_\_l2h\_gamma=2.2f, stbi\_\_l2h\_scale=1.0f;

STBIDEF void stbi\_ldr\_to\_hdr\_gamma(float gamma) { stbi\_\_l2h\_gamma = gamma; }

STBIDEF void stbi\_ldr\_to\_hdr\_scale(float scale) { stbi\_\_l2h\_scale = scale; }

#endif

static float stbi\_\_h2l\_gamma\_i=1.0f/2.2f, stbi\_\_h2l\_scale\_i=1.0f;

STBIDEF void stbi\_hdr\_to\_ldr\_gamma(float gamma) { stbi\_\_h2l\_gamma\_i = 1/gamma; }

STBIDEF void stbi\_hdr\_to\_ldr\_scale(float scale) { stbi\_\_h2l\_scale\_i = 1/scale; }

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

//

// Common code used by all image loaders

//

enum

{

STBI\_\_SCAN\_load=0,

STBI\_\_SCAN\_type,

STBI\_\_SCAN\_header

};

static void stbi\_\_refill\_buffer(stbi\_\_context \*s)

{

int n = (s->io.read)(s->io\_user\_data,(char\*)s->buffer\_start,s->buflen);

if (n == 0) {

// at end of file, treat same as if from memory, but need to handle case

// where s->img\_buffer isn't pointing to safe memory, e.g. 0-byte file

s->read\_from\_callbacks = 0;

s->img\_buffer = s->buffer\_start;

s->img\_buffer\_end = s->buffer\_start+1;

\*s->img\_buffer = 0;

} else {

s->img\_buffer = s->buffer\_start;

s->img\_buffer\_end = s->buffer\_start + n;

}

}

stbi\_inline static stbi\_uc stbi\_\_get8(stbi\_\_context \*s)

{

if (s->img\_buffer < s->img\_buffer\_end)

return \*s->img\_buffer++;

if (s->read\_from\_callbacks) {

stbi\_\_refill\_buffer(s);

return \*s->img\_buffer++;

}

return 0;

}

stbi\_inline static int stbi\_\_at\_eof(stbi\_\_context \*s)

{

if (s->io.read) {

if (!(s->io.eof)(s->io\_user\_data)) return 0;

// if feof() is true, check if buffer = end

// special case: we've only got the special 0 character at the end

if (s->read\_from\_callbacks == 0) return 1;

}

return s->img\_buffer >= s->img\_buffer\_end;

}

static void stbi\_\_skip(stbi\_\_context \*s, int n)

{

if (n < 0) {

s->img\_buffer = s->img\_buffer\_end;

return;

}

if (s->io.read) {

int blen = (int) (s->img\_buffer\_end - s->img\_buffer);

if (blen < n) {

s->img\_buffer = s->img\_buffer\_end;

(s->io.skip)(s->io\_user\_data, n - blen);

return;

}

}

s->img\_buffer += n;

}

static int stbi\_\_getn(stbi\_\_context \*s, stbi\_uc \*buffer, int n)

{

if (s->io.read) {

int blen = (int) (s->img\_buffer\_end - s->img\_buffer);

if (blen < n) {

int res, count;

memcpy(buffer, s->img\_buffer, blen);

count = (s->io.read)(s->io\_user\_data, (char\*) buffer + blen, n - blen);

res = (count == (n-blen));

s->img\_buffer = s->img\_buffer\_end;

return res;

}

}

if (s->img\_buffer+n <= s->img\_buffer\_end) {

memcpy(buffer, s->img\_buffer, n);

s->img\_buffer += n;

return 1;

} else

return 0;

}

static int stbi\_\_get16be(stbi\_\_context \*s)

{

int z = stbi\_\_get8(s);

return (z << 8) + stbi\_\_get8(s);

}

static stbi\_\_uint32 stbi\_\_get32be(stbi\_\_context \*s)

{

stbi\_\_uint32 z = stbi\_\_get16be(s);

return (z << 16) + stbi\_\_get16be(s);

}

#if defined(STBI\_NO\_BMP) && defined(STBI\_NO\_TGA) && defined(STBI\_NO\_GIF)

// nothing

#else

static int stbi\_\_get16le(stbi\_\_context \*s)

{

int z = stbi\_\_get8(s);

return z + (stbi\_\_get8(s) << 8);

}

#endif

#ifndef STBI\_NO\_BMP

static stbi\_\_uint32 stbi\_\_get32le(stbi\_\_context \*s)

{

stbi\_\_uint32 z = stbi\_\_get16le(s);

return z + (stbi\_\_get16le(s) << 16);

}

#endif

#define STBI\_\_BYTECAST(x) ((stbi\_uc) ((x) & 255)) // truncate int to byte without warnings

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

//

// generic converter from built-in img\_n to req\_comp

// individual types do this automatically as much as possible (e.g. jpeg

// does all cases internally since it needs to colorspace convert anyway,

// and it never has alpha, so very few cases ). png can automatically

// interleave an alpha=255 channel, but falls back to this for other cases

//

// assume data buffer is malloced, so malloc a new one and free that one

// only failure mode is malloc failing

static stbi\_uc stbi\_\_compute\_y(int r, int g, int b)

{

return (stbi\_uc) (((r\*77) + (g\*150) + (29\*b)) >> 8);

}

static unsigned char \*stbi\_\_convert\_format(unsigned char \*data, int img\_n, int req\_comp, unsigned int x, unsigned int y)

{

int i,j;

unsigned char \*good;

if (req\_comp == img\_n) return data;

STBI\_ASSERT(req\_comp >= 1 && req\_comp <= 4);

good = (unsigned char \*) stbi\_\_malloc\_mad3(req\_comp, x, y, 0);

if (good == NULL) {

STBI\_FREE(data);

return stbi\_\_errpuc("outofmem", "Out of memory");

}

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

unsigned char \*src = data + j \* x \* img\_n ;

unsigned char \*dest = good + j \* x \* req\_comp;

#define STBI\_\_COMBO(a,b) ((a)\*8+(b))

#define STBI\_\_CASE(a,b) case STBI\_\_COMBO(a,b): for(i=x-1; i >= 0; --i, src += a, dest += b)

// convert source image with img\_n components to one with req\_comp components;

// avoid switch per pixel, so use switch per scanline and massive macros

switch (STBI\_\_COMBO(img\_n, req\_comp)) {

STBI\_\_CASE(1,2) { dest[0]=src[0], dest[1]=255; } break;

STBI\_\_CASE(1,3) { dest[0]=dest[1]=dest[2]=src[0]; } break;

STBI\_\_CASE(1,4) { dest[0]=dest[1]=dest[2]=src[0], dest[3]=255; } break;

STBI\_\_CASE(2,1) { dest[0]=src[0]; } break;

STBI\_\_CASE(2,3) { dest[0]=dest[1]=dest[2]=src[0]; } break;

STBI\_\_CASE(2,4) { dest[0]=dest[1]=dest[2]=src[0], dest[3]=src[1]; } break;

STBI\_\_CASE(3,4) { dest[0]=src[0],dest[1]=src[1],dest[2]=src[2],dest[3]=255; } break;

STBI\_\_CASE(3,1) { dest[0]=stbi\_\_compute\_y(src[0],src[1],src[2]); } break;

STBI\_\_CASE(3,2) { dest[0]=stbi\_\_compute\_y(src[0],src[1],src[2]), dest[1] = 255; } break;

STBI\_\_CASE(4,1) { dest[0]=stbi\_\_compute\_y(src[0],src[1],src[2]); } break;

STBI\_\_CASE(4,2) { dest[0]=stbi\_\_compute\_y(src[0],src[1],src[2]), dest[1] = src[3]; } break;

STBI\_\_CASE(4,3) { dest[0]=src[0],dest[1]=src[1],dest[2]=src[2]; } break;

default: STBI\_ASSERT(0);

}

#undef STBI\_\_CASE

}

STBI\_FREE(data);

return good;

}

static stbi\_\_uint16 stbi\_\_compute\_y\_16(int r, int g, int b)

{

return (stbi\_\_uint16) (((r\*77) + (g\*150) + (29\*b)) >> 8);

}

static stbi\_\_uint16 \*stbi\_\_convert\_format16(stbi\_\_uint16 \*data, int img\_n, int req\_comp, unsigned int x, unsigned int y)

{

int i,j;

stbi\_\_uint16 \*good;

if (req\_comp == img\_n) return data;

STBI\_ASSERT(req\_comp >= 1 && req\_comp <= 4);

good = (stbi\_\_uint16 \*) stbi\_\_malloc(req\_comp \* x \* y \* 2);

if (good == NULL) {

STBI\_FREE(data);

return (stbi\_\_uint16 \*) stbi\_\_errpuc("outofmem", "Out of memory");

}

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

stbi\_\_uint16 \*src = data + j \* x \* img\_n ;

stbi\_\_uint16 \*dest = good + j \* x \* req\_comp;

#define STBI\_\_COMBO(a,b) ((a)\*8+(b))

#define STBI\_\_CASE(a,b) case STBI\_\_COMBO(a,b): for(i=x-1; i >= 0; --i, src += a, dest += b)

// convert source image with img\_n components to one with req\_comp components;

// avoid switch per pixel, so use switch per scanline and massive macros

switch (STBI\_\_COMBO(img\_n, req\_comp)) {

STBI\_\_CASE(1,2) { dest[0]=src[0], dest[1]=0xffff; } break;

STBI\_\_CASE(1,3) { dest[0]=dest[1]=dest[2]=src[0]; } break;

STBI\_\_CASE(1,4) { dest[0]=dest[1]=dest[2]=src[0], dest[3]=0xffff; } break;

STBI\_\_CASE(2,1) { dest[0]=src[0]; } break;

STBI\_\_CASE(2,3) { dest[0]=dest[1]=dest[2]=src[0]; } break;

STBI\_\_CASE(2,4) { dest[0]=dest[1]=dest[2]=src[0], dest[3]=src[1]; } break;

STBI\_\_CASE(3,4) { dest[0]=src[0],dest[1]=src[1],dest[2]=src[2],dest[3]=0xffff; } break;

STBI\_\_CASE(3,1) { dest[0]=stbi\_\_compute\_y\_16(src[0],src[1],src[2]); } break;

STBI\_\_CASE(3,2) { dest[0]=stbi\_\_compute\_y\_16(src[0],src[1],src[2]), dest[1] = 0xffff; } break;

STBI\_\_CASE(4,1) { dest[0]=stbi\_\_compute\_y\_16(src[0],src[1],src[2]); } break;

STBI\_\_CASE(4,2) { dest[0]=stbi\_\_compute\_y\_16(src[0],src[1],src[2]), dest[1] = src[3]; } break;

STBI\_\_CASE(4,3) { dest[0]=src[0],dest[1]=src[1],dest[2]=src[2]; } break;

default: STBI\_ASSERT(0);

}

#undef STBI\_\_CASE

}

STBI\_FREE(data);

return good;

}

#ifndef STBI\_NO\_LINEAR

static float \*stbi\_\_ldr\_to\_hdr(stbi\_uc \*data, int x, int y, int comp)

{

int i,k,n;

float \*output;

if (!data) return NULL;

output = (float \*) stbi\_\_malloc\_mad4(x, y, comp, sizeof(float), 0);

if (output == NULL) { STBI\_FREE(data); return stbi\_\_errpf("outofmem", "Out of memory"); }

// compute number of non-alpha components

if (comp & 1) n = comp; else n = comp-1;

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

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

output[i\*comp + k] = (float) (pow(data[i\*comp+k]/255.0f, stbi\_\_l2h\_gamma) \* stbi\_\_l2h\_scale);

}

if (k < comp) output[i\*comp + k] = data[i\*comp+k]/255.0f;

}

STBI\_FREE(data);

return output;

}

#endif

#ifndef STBI\_NO\_HDR

#define stbi\_\_float2int(x) ((int) (x))

static stbi\_uc \*stbi\_\_hdr\_to\_ldr(float \*data, int x, int y, int comp)

{

int i,k,n;

stbi\_uc \*output;

if (!data) return NULL;

output = (stbi\_uc \*) stbi\_\_malloc\_mad3(x, y, comp, 0);

if (output == NULL) { STBI\_FREE(data); return stbi\_\_errpuc("outofmem", "Out of memory"); }

// compute number of non-alpha components

if (comp & 1) n = comp; else n = comp-1;

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

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

float z = (float) pow(data[i\*comp+k]\*stbi\_\_h2l\_scale\_i, stbi\_\_h2l\_gamma\_i) \* 255 + 0.5f;

if (z < 0) z = 0;

if (z > 255) z = 255;

output[i\*comp + k] = (stbi\_uc) stbi\_\_float2int(z);

}

if (k < comp) {

float z = data[i\*comp+k] \* 255 + 0.5f;

if (z < 0) z = 0;

if (z > 255) z = 255;

output[i\*comp + k] = (stbi\_uc) stbi\_\_float2int(z);

}

}

STBI\_FREE(data);

return output;

}

#endif

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

//

// "baseline" JPEG/JFIF decoder

//

// simple implementation

// - doesn't support delayed output of y-dimension

// - simple interface (only one output format: 8-bit interleaved RGB)

// - doesn't try to recover corrupt jpegs

// - doesn't allow partial loading, loading multiple at once

// - still fast on x86 (copying globals into locals doesn't help x86)

// - allocates lots of intermediate memory (full size of all components)

// - non-interleaved case requires this anyway

// - allows good upsampling (see next)

// high-quality

// - upsampled channels are bilinearly interpolated, even across blocks

// - quality integer IDCT derived from IJG's 'slow'

// performance

// - fast huffman; reasonable integer IDCT

// - some SIMD kernels for common paths on targets with SSE2/NEON

// - uses a lot of intermediate memory, could cache poorly

#ifndef STBI\_NO\_JPEG

// huffman decoding acceleration

#define FAST\_BITS 9 // larger handles more cases; smaller stomps less cache

typedef struct

{

stbi\_uc fast[1 << FAST\_BITS];

// weirdly, repacking this into AoS is a 10% speed loss, instead of a win

stbi\_\_uint16 code[256];

stbi\_uc values[256];

stbi\_uc size[257];

unsigned int maxcode[18];

int delta[17]; // old 'firstsymbol' - old 'firstcode'

} stbi\_\_huffman;

typedef struct

{

stbi\_\_context \*s;

stbi\_\_huffman huff\_dc[4];

stbi\_\_huffman huff\_ac[4];

stbi\_\_uint16 dequant[4][64];

stbi\_\_int16 fast\_ac[4][1 << FAST\_BITS];

// sizes for components, interleaved MCUs

int img\_h\_max, img\_v\_max;

int img\_mcu\_x, img\_mcu\_y;

int img\_mcu\_w, img\_mcu\_h;

// definition of jpeg image component

struct

{

int id;

int h,v;

int tq;

int hd,ha;

int dc\_pred;

int x,y,w2,h2;

stbi\_uc \*data;

void \*raw\_data, \*raw\_coeff;

stbi\_uc \*linebuf;

short \*coeff; // progressive only

int coeff\_w, coeff\_h; // number of 8x8 coefficient blocks

} img\_comp[4];

stbi\_\_uint32 code\_buffer; // jpeg entropy-coded buffer

int code\_bits; // number of valid bits

unsigned char marker; // marker seen while filling entropy buffer

int nomore; // flag if we saw a marker so must stop

int progressive;

int spec\_start;

int spec\_end;

int succ\_high;

int succ\_low;

int eob\_run;

int jfif;

int app14\_color\_transform; // Adobe APP14 tag

int rgb;

int scan\_n, order[4];

int restart\_interval, todo;

// kernels

void (\*idct\_block\_kernel)(stbi\_uc \*out, int out\_stride, short data[64]);

void (\*YCbCr\_to\_RGB\_kernel)(stbi\_uc \*out, const stbi\_uc \*y, const stbi\_uc \*pcb, const stbi\_uc \*pcr, int count, int step);

stbi\_uc \*(\*resample\_row\_hv\_2\_kernel)(stbi\_uc \*out, stbi\_uc \*in\_near, stbi\_uc \*in\_far, int w, int hs);

} stbi\_\_jpeg;

static int stbi\_\_build\_huffman(stbi\_\_huffman \*h, int \*count)

{

int i,j,k=0,code;

// build size list for each symbol (from JPEG spec)

for (i=0; i < 16; ++i)

for (j=0; j < count[i]; ++j)

h->size[k++] = (stbi\_uc) (i+1);

h->size[k] = 0;

// compute actual symbols (from jpeg spec)

code = 0;

k = 0;

for(j=1; j <= 16; ++j) {

// compute delta to add to code to compute symbol id

h->delta[j] = k - code;

if (h->size[k] == j) {

while (h->size[k] == j)

h->code[k++] = (stbi\_\_uint16) (code++);

if (code-1 >= (1 << j)) return stbi\_\_err("bad code lengths","Corrupt JPEG");

}

// compute largest code + 1 for this size, preshifted as needed later

h->maxcode[j] = code << (16-j);

code <<= 1;

}

h->maxcode[j] = 0xffffffff;

// build non-spec acceleration table; 255 is flag for not-accelerated

memset(h->fast, 255, 1 << FAST\_BITS);

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

int s = h->size[i];

if (s <= FAST\_BITS) {

int c = h->code[i] << (FAST\_BITS-s);

int m = 1 << (FAST\_BITS-s);

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

h->fast[c+j] = (stbi\_uc) i;

}

}

}

return 1;

}

// build a table that decodes both magnitude and value of small ACs in

// one go.

static void stbi\_\_build\_fast\_ac(stbi\_\_int16 \*fast\_ac, stbi\_\_huffman \*h)

{

int i;

for (i=0; i < (1 << FAST\_BITS); ++i) {

stbi\_uc fast = h->fast[i];

fast\_ac[i] = 0;

if (fast < 255) {

int rs = h->values[fast];

int run = (rs >> 4) & 15;

int magbits = rs & 15;

int len = h->size[fast];

if (magbits && len + magbits <= FAST\_BITS) {

// magnitude code followed by receive\_extend code

int k = ((i << len) & ((1 << FAST\_BITS) - 1)) >> (FAST\_BITS - magbits);

int m = 1 << (magbits - 1);

if (k < m) k += (~0U << magbits) + 1;

// if the result is small enough, we can fit it in fast\_ac table

if (k >= -128 && k <= 127)

fast\_ac[i] = (stbi\_\_int16) ((k << 8) + (run << 4) + (len + magbits));

}

}

}

}

static void stbi\_\_grow\_buffer\_unsafe(stbi\_\_jpeg \*j)

{

do {

int b = j->nomore ? 0 : stbi\_\_get8(j->s);

if (b == 0xff) {

int c = stbi\_\_get8(j->s);

while (c == 0xff) c = stbi\_\_get8(j->s); // consume fill bytes

if (c != 0) {

j->marker = (unsigned char) c;

j->nomore = 1;

return;

}

}

j->code\_buffer |= b << (24 - j->code\_bits);

j->code\_bits += 8;

} while (j->code\_bits <= 24);

}

// (1 << n) - 1

static stbi\_\_uint32 stbi\_\_bmask[17]={0,1,3,7,15,31,63,127,255,511,1023,2047,4095,8191,16383,32767,65535};

// decode a jpeg huffman value from the bitstream

stbi\_inline static int stbi\_\_jpeg\_huff\_decode(stbi\_\_jpeg \*j, stbi\_\_huffman \*h)

{

unsigned int temp;

int c,k;

if (j->code\_bits < 16) stbi\_\_grow\_buffer\_unsafe(j);

// look at the top FAST\_BITS and determine what symbol ID it is,

// if the code is <= FAST\_BITS

c = (j->code\_buffer >> (32 - FAST\_BITS)) & ((1 << FAST\_BITS)-1);

k = h->fast[c];

if (k < 255) {

int s = h->size[k];

if (s > j->code\_bits)

return -1;

j->code\_buffer <<= s;

j->code\_bits -= s;

return h->values[k];

}

// naive test is to shift the code\_buffer down so k bits are

// valid, then test against maxcode. To speed this up, we've

// preshifted maxcode left so that it has (16-k) 0s at the

// end; in other words, regardless of the number of bits, it

// wants to be compared against something shifted to have 16;

// that way we don't need to shift inside the loop.

temp = j->code\_buffer >> 16;

for (k=FAST\_BITS+1 ; ; ++k)

if (temp < h->maxcode[k])

break;

if (k == 17) {

// error! code not found

j->code\_bits -= 16;

return -1;

}

if (k > j->code\_bits)

return -1;

// convert the huffman code to the symbol id

c = ((j->code\_buffer >> (32 - k)) & stbi\_\_bmask[k]) + h->delta[k];

STBI\_ASSERT((((j->code\_buffer) >> (32 - h->size[c])) & stbi\_\_bmask[h->size[c]]) == h->code[c]);

// convert the id to a symbol

j->code\_bits -= k;

j->code\_buffer <<= k;

return h->values[c];

}

// bias[n] = (-1<<n) + 1

static int const stbi\_\_jbias[16] = {0,-1,-3,-7,-15,-31,-63,-127,-255,-511,-1023,-2047,-4095,-8191,-16383,-32767};

// combined JPEG 'receive' and JPEG 'extend', since baseline

// always extends everything it receives.

stbi\_inline static int stbi\_\_extend\_receive(stbi\_\_jpeg \*j, int n)

{

unsigned int k;

int sgn;

if (j->code\_bits < n) stbi\_\_grow\_buffer\_unsafe(j);

sgn = (stbi\_\_int32)j->code\_buffer >> 31; // sign bit is always in MSB

k = stbi\_lrot(j->code\_buffer, n);

STBI\_ASSERT(n >= 0 && n < (int) (sizeof(stbi\_\_bmask)/sizeof(\*stbi\_\_bmask)));

j->code\_buffer = k & ~stbi\_\_bmask[n];

k &= stbi\_\_bmask[n];

j->code\_bits -= n;

return k + (stbi\_\_jbias[n] & ~sgn);

}

// get some unsigned bits

stbi\_inline static int stbi\_\_jpeg\_get\_bits(stbi\_\_jpeg \*j, int n)

{

unsigned int k;

if (j->code\_bits < n) stbi\_\_grow\_buffer\_unsafe(j);

k = stbi\_lrot(j->code\_buffer, n);

j->code\_buffer = k & ~stbi\_\_bmask[n];

k &= stbi\_\_bmask[n];

j->code\_bits -= n;

return k;

}

stbi\_inline static int stbi\_\_jpeg\_get\_bit(stbi\_\_jpeg \*j)

{

unsigned int k;

if (j->code\_bits < 1) stbi\_\_grow\_buffer\_unsafe(j);

k = j->code\_buffer;

j->code\_buffer <<= 1;

--j->code\_bits;

return k & 0x80000000;

}

// given a value that's at position X in the zigzag stream,

// where does it appear in the 8x8 matrix coded as row-major?

static stbi\_uc stbi\_\_jpeg\_dezigzag[64+15] =

{

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,

// let corrupt input sample past end

63, 63, 63, 63, 63, 63, 63, 63,

63, 63, 63, 63, 63, 63, 63

};

// decode one 64-entry block--

static int stbi\_\_jpeg\_decode\_block(stbi\_\_jpeg \*j, short data[64], stbi\_\_huffman \*hdc, stbi\_\_huffman \*hac, stbi\_\_int16 \*fac, int b, stbi\_\_uint16 \*dequant)

{

int diff,dc,k;

int t;

if (j->code\_bits < 16) stbi\_\_grow\_buffer\_unsafe(j);

t = stbi\_\_jpeg\_huff\_decode(j, hdc);

if (t < 0) return stbi\_\_err("bad huffman code","Corrupt JPEG");

// 0 all the ac values now so we can do it 32-bits at a time

memset(data,0,64\*sizeof(data[0]));

diff = t ? stbi\_\_extend\_receive(j, t) : 0;

dc = j->img\_comp[b].dc\_pred + diff;

j->img\_comp[b].dc\_pred = dc;

data[0] = (short) (dc \* dequant[0]);

// decode AC components, see JPEG spec

k = 1;

do {

unsigned int zig;

int c,r,s;

if (j->code\_bits < 16) stbi\_\_grow\_buffer\_unsafe(j);

c = (j->code\_buffer >> (32 - FAST\_BITS)) & ((1 << FAST\_BITS)-1);

r = fac[c];

if (r) { // fast-AC path

k += (r >> 4) & 15; // run

s = r & 15; // combined length

j->code\_buffer <<= s;

j->code\_bits -= s;

// decode into unzigzag'd location

zig = stbi\_\_jpeg\_dezigzag[k++];

data[zig] = (short) ((r >> 8) \* dequant[zig]);

} else {

int rs = stbi\_\_jpeg\_huff\_decode(j, hac);

if (rs < 0) return stbi\_\_err("bad huffman code","Corrupt JPEG");

s = rs & 15;

r = rs >> 4;

if (s == 0) {

if (rs != 0xf0) break; // end block

k += 16;

} else {

k += r;

// decode into unzigzag'd location

zig = stbi\_\_jpeg\_dezigzag[k++];

data[zig] = (short) (stbi\_\_extend\_receive(j,s) \* dequant[zig]);

}

}

} while (k < 64);

return 1;

}

static int stbi\_\_jpeg\_decode\_block\_prog\_dc(stbi\_\_jpeg \*j, short data[64], stbi\_\_huffman \*hdc, int b)

{

int diff,dc;

int t;

if (j->spec\_end != 0) return stbi\_\_err("can't merge dc and ac", "Corrupt JPEG");

if (j->code\_bits < 16) stbi\_\_grow\_buffer\_unsafe(j);

if (j->succ\_high == 0) {

// first scan for DC coefficient, must be first

memset(data,0,64\*sizeof(data[0])); // 0 all the ac values now

t = stbi\_\_jpeg\_huff\_decode(j, hdc);

diff = t ? stbi\_\_extend\_receive(j, t) : 0;

dc = j->img\_comp[b].dc\_pred + diff;

j->img\_comp[b].dc\_pred = dc;

data[0] = (short) (dc << j->succ\_low);

} else {

// refinement scan for DC coefficient

if (stbi\_\_jpeg\_get\_bit(j))

data[0] += (short) (1 << j->succ\_low);

}

return 1;

}

// @OPTIMIZE: store non-zigzagged during the decode passes,

// and only de-zigzag when dequantizing

static int stbi\_\_jpeg\_decode\_block\_prog\_ac(stbi\_\_jpeg \*j, short data[64], stbi\_\_huffman \*hac, stbi\_\_int16 \*fac)

{

int k;

if (j->spec\_start == 0) return stbi\_\_err("can't merge dc and ac", "Corrupt JPEG");

if (j->succ\_high == 0) {

int shift = j->succ\_low;

if (j->eob\_run) {

--j->eob\_run;

return 1;

}

k = j->spec\_start;

do {

unsigned int zig;

int c,r,s;

if (j->code\_bits < 16) stbi\_\_grow\_buffer\_unsafe(j);

c = (j->code\_buffer >> (32 - FAST\_BITS)) & ((1 << FAST\_BITS)-1);

r = fac[c];

if (r) { // fast-AC path

k += (r >> 4) & 15; // run

s = r & 15; // combined length

j->code\_buffer <<= s;

j->code\_bits -= s;

zig = stbi\_\_jpeg\_dezigzag[k++];

data[zig] = (short) ((r >> 8) << shift);

} else {

int rs = stbi\_\_jpeg\_huff\_decode(j, hac);

if (rs < 0) return stbi\_\_err("bad huffman code","Corrupt JPEG");

s = rs & 15;

r = rs >> 4;

if (s == 0) {

if (r < 15) {

j->eob\_run = (1 << r);

if (r)

j->eob\_run += stbi\_\_jpeg\_get\_bits(j, r);

--j->eob\_run;

break;

}

k += 16;

} else {

k += r;

zig = stbi\_\_jpeg\_dezigzag[k++];

data[zig] = (short) (stbi\_\_extend\_receive(j,s) << shift);

}

}

} while (k <= j->spec\_end);

} else {

// refinement scan for these AC coefficients

short bit = (short) (1 << j->succ\_low);

if (j->eob\_run) {

--j->eob\_run;

for (k = j->spec\_start; k <= j->spec\_end; ++k) {

short \*p = &data[stbi\_\_jpeg\_dezigzag[k]];

if (\*p != 0)

if (stbi\_\_jpeg\_get\_bit(j))

if ((\*p & bit)==0) {

if (\*p > 0)

\*p += bit;

else

\*p -= bit;

}

}

} else {

k = j->spec\_start;

do {

int r,s;

int rs = stbi\_\_jpeg\_huff\_decode(j, hac); // @OPTIMIZE see if we can use the fast path here, advance-by-r is so slow, eh

if (rs < 0) return stbi\_\_err("bad huffman code","Corrupt JPEG");

s = rs & 15;

r = rs >> 4;

if (s == 0) {

if (r < 15) {

j->eob\_run = (1 << r) - 1;

if (r)

j->eob\_run += stbi\_\_jpeg\_get\_bits(j, r);

r = 64; // force end of block

} else {

// r=15 s=0 should write 16 0s, so we just do

// a run of 15 0s and then write s (which is 0),

// so we don't have to do anything special here

}

} else {

if (s != 1) return stbi\_\_err("bad huffman code", "Corrupt JPEG");

// sign bit

if (stbi\_\_jpeg\_get\_bit(j))

s = bit;

else

s = -bit;

}

// advance by r

while (k <= j->spec\_end) {

short \*p = &data[stbi\_\_jpeg\_dezigzag[k++]];

if (\*p != 0) {

if (stbi\_\_jpeg\_get\_bit(j))

if ((\*p & bit)==0) {

if (\*p > 0)

\*p += bit;

else

\*p -= bit;

}

} else {

if (r == 0) {

\*p = (short) s;

break;

}

--r;

}

}

} while (k <= j->spec\_end);

}

}

return 1;

}

// take a -128..127 value and stbi\_\_clamp it and convert to 0..255

stbi\_inline static stbi\_uc stbi\_\_clamp(int x)

{

// trick to use a single test to catch both cases

if ((unsigned int) x > 255) {

if (x < 0) return 0;

if (x > 255) return 255;

}

return (stbi\_uc) x;

}

#define stbi\_\_f2f(x) ((int) (((x) \* 4096 + 0.5)))

#define stbi\_\_fsh(x) ((x) << 12)

// derived from jidctint -- DCT\_ISLOW

#define STBI\_\_IDCT\_1D(s0,s1,s2,s3,s4,s5,s6,s7) \

int t0,t1,t2,t3,p1,p2,p3,p4,p5,x0,x1,x2,x3; \

p2 = s2; \

p3 = s6; \

p1 = (p2+p3) \* stbi\_\_f2f(0.5411961f); \

t2 = p1 + p3\*stbi\_\_f2f(-1.847759065f); \

t3 = p1 + p2\*stbi\_\_f2f( 0.765366865f); \

p2 = s0; \

p3 = s4; \

t0 = stbi\_\_fsh(p2+p3); \

t1 = stbi\_\_fsh(p2-p3); \

x0 = t0+t3; \

x3 = t0-t3; \

x1 = t1+t2; \

x2 = t1-t2; \

t0 = s7; \

t1 = s5; \

t2 = s3; \

t3 = s1; \

p3 = t0+t2; \

p4 = t1+t3; \

p1 = t0+t3; \

p2 = t1+t2; \

p5 = (p3+p4)\*stbi\_\_f2f( 1.175875602f); \

t0 = t0\*stbi\_\_f2f( 0.298631336f); \

t1 = t1\*stbi\_\_f2f( 2.053119869f); \

t2 = t2\*stbi\_\_f2f( 3.072711026f); \

t3 = t3\*stbi\_\_f2f( 1.501321110f); \

p1 = p5 + p1\*stbi\_\_f2f(-0.899976223f); \

p2 = p5 + p2\*stbi\_\_f2f(-2.562915447f); \

p3 = p3\*stbi\_\_f2f(-1.961570560f); \

p4 = p4\*stbi\_\_f2f(-0.390180644f); \

t3 += p1+p4; \

t2 += p2+p3; \

t1 += p2+p4; \

t0 += p1+p3;

static void stbi\_\_idct\_block(stbi\_uc \*out, int out\_stride, short data[64])

{

int i,val[64],\*v=val;

stbi\_uc \*o;

short \*d = data;

// columns

for (i=0; i < 8; ++i,++d, ++v) {

// if all zeroes, shortcut -- this avoids dequantizing 0s and IDCTing

if (d[ 8]==0 && d[16]==0 && d[24]==0 && d[32]==0

&& d[40]==0 && d[48]==0 && d[56]==0) {

// no shortcut 0 seconds

// (1|2|3|4|5|6|7)==0 0 seconds

// all separate -0.047 seconds

// 1 && 2|3 && 4|5 && 6|7: -0.047 seconds

int dcterm = d[0] << 2;

v[0] = v[8] = v[16] = v[24] = v[32] = v[40] = v[48] = v[56] = dcterm;

} else {

STBI\_\_IDCT\_1D(d[ 0],d[ 8],d[16],d[24],d[32],d[40],d[48],d[56])

// constants scaled things up by 1<<12; let's bring them back

// down, but keep 2 extra bits of precision

x0 += 512; x1 += 512; x2 += 512; x3 += 512;

v[ 0] = (x0+t3) >> 10;

v[56] = (x0-t3) >> 10;

v[ 8] = (x1+t2) >> 10;

v[48] = (x1-t2) >> 10;

v[16] = (x2+t1) >> 10;

v[40] = (x2-t1) >> 10;

v[24] = (x3+t0) >> 10;

v[32] = (x3-t0) >> 10;

}

}

for (i=0, v=val, o=out; i < 8; ++i,v+=8,o+=out\_stride) {

// no fast case since the first 1D IDCT spread components out

STBI\_\_IDCT\_1D(v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7])

// constants scaled things up by 1<<12, plus we had 1<<2 from first

// loop, plus horizontal and vertical each scale by sqrt(8) so together

// we've got an extra 1<<3, so 1<<17 total we need to remove.

// so we want to round that, which means adding 0.5 \* 1<<17,

// aka 65536. Also, we'll end up with -128 to 127 that we want

// to encode as 0..255 by adding 128, so we'll add that before the shift

x0 += 65536 + (128<<17);

x1 += 65536 + (128<<17);

x2 += 65536 + (128<<17);

x3 += 65536 + (128<<17);

// tried computing the shifts into temps, or'ing the temps to see

// if any were out of range, but that was slower

o[0] = stbi\_\_clamp((x0+t3) >> 17);

o[7] = stbi\_\_clamp((x0-t3) >> 17);

o[1] = stbi\_\_clamp((x1+t2) >> 17);

o[6] = stbi\_\_clamp((x1-t2) >> 17);

o[2] = stbi\_\_clamp((x2+t1) >> 17);

o[5] = stbi\_\_clamp((x2-t1) >> 17);

o[3] = stbi\_\_clamp((x3+t0) >> 17);

o[4] = stbi\_\_clamp((x3-t0) >> 17);

}

}

#ifdef STBI\_SSE2

// sse2 integer IDCT. not the fastest possible implementation but it

// produces bit-identical results to the generic C version so it's

// fully "transparent".

static void stbi\_\_idct\_simd(stbi\_uc \*out, int out\_stride, short data[64])

{

// This is constructed to match our regular (generic) integer IDCT exactly.

\_\_m128i row0, row1, row2, row3, row4, row5, row6, row7;

\_\_m128i tmp;

// dot product constant: even elems=x, odd elems=y

#define dct\_const(x,y) \_mm\_setr\_epi16((x),(y),(x),(y),(x),(y),(x),(y))

// out(0) = c0[even]\*x + c0[odd]\*y (c0, x, y 16-bit, out 32-bit)

// out(1) = c1[even]\*x + c1[odd]\*y

#define dct\_rot(out0,out1, x,y,c0,c1) \

\_\_m128i c0##lo = \_mm\_unpacklo\_epi16((x),(y)); \

\_\_m128i c0##hi = \_mm\_unpackhi\_epi16((x),(y)); \

\_\_m128i out0##\_l = \_mm\_madd\_epi16(c0##lo, c0); \

\_\_m128i out0##\_h = \_mm\_madd\_epi16(c0##hi, c0); \

\_\_m128i out1##\_l = \_mm\_madd\_epi16(c0##lo, c1); \

\_\_m128i out1##\_h = \_mm\_madd\_epi16(c0##hi, c1)

// out = in << 12 (in 16-bit, out 32-bit)

#define dct\_widen(out, in) \

\_\_m128i out##\_l = \_mm\_srai\_epi32(\_mm\_unpacklo\_epi16(\_mm\_setzero\_si128(), (in)), 4); \

\_\_m128i out##\_h = \_mm\_srai\_epi32(\_mm\_unpackhi\_epi16(\_mm\_setzero\_si128(), (in)), 4)

// wide add

#define dct\_wadd(out, a, b) \

\_\_m128i out##\_l = \_mm\_add\_epi32(a##\_l, b##\_l); \

\_\_m128i out##\_h = \_mm\_add\_epi32(a##\_h, b##\_h)

// wide sub

#define dct\_wsub(out, a, b) \

\_\_m128i out##\_l = \_mm\_sub\_epi32(a##\_l, b##\_l); \

\_\_m128i out##\_h = \_mm\_sub\_epi32(a##\_h, b##\_h)

// butterfly a/b, add bias, then shift by "s" and pack

#define dct\_bfly32o(out0, out1, a,b,bias,s) \

{ \

\_\_m128i abiased\_l = \_mm\_add\_epi32(a##\_l, bias); \

\_\_m128i abiased\_h = \_mm\_add\_epi32(a##\_h, bias); \

dct\_wadd(sum, abiased, b); \

dct\_wsub(dif, abiased, b); \

out0 = \_mm\_packs\_epi32(\_mm\_srai\_epi32(sum\_l, s), \_mm\_srai\_epi32(sum\_h, s)); \

out1 = \_mm\_packs\_epi32(\_mm\_srai\_epi32(dif\_l, s), \_mm\_srai\_epi32(dif\_h, s)); \

}

// 8-bit interleave step (for transposes)

#define dct\_interleave8(a, b) \

tmp = a; \

a = \_mm\_unpacklo\_epi8(a, b); \

b = \_mm\_unpackhi\_epi8(tmp, b)

// 16-bit interleave step (for transposes)

#define dct\_interleave16(a, b) \

tmp = a; \

a = \_mm\_unpacklo\_epi16(a, b); \

b = \_mm\_unpackhi\_epi16(tmp, b)

#define dct\_pass(bias,shift) \

{ \

/\* even part \*/ \

dct\_rot(t2e,t3e, row2,row6, rot0\_0,rot0\_1); \

\_\_m128i sum04 = \_mm\_add\_epi16(row0, row4); \

\_\_m128i dif04 = \_mm\_sub\_epi16(row0, row4); \

dct\_widen(t0e, sum04); \

dct\_widen(t1e, dif04); \

dct\_wadd(x0, t0e, t3e); \

dct\_wsub(x3, t0e, t3e); \

dct\_wadd(x1, t1e, t2e); \

dct\_wsub(x2, t1e, t2e); \

/\* odd part \*/ \

dct\_rot(y0o,y2o, row7,row3, rot2\_0,rot2\_1); \

dct\_rot(y1o,y3o, row5,row1, rot3\_0,rot3\_1); \

\_\_m128i sum17 = \_mm\_add\_epi16(row1, row7); \

\_\_m128i sum35 = \_mm\_add\_epi16(row3, row5); \

dct\_rot(y4o,y5o, sum17,sum35, rot1\_0,rot1\_1); \

dct\_wadd(x4, y0o, y4o); \

dct\_wadd(x5, y1o, y5o); \

dct\_wadd(x6, y2o, y5o); \

dct\_wadd(x7, y3o, y4o); \

dct\_bfly32o(row0,row7, x0,x7,bias,shift); \

dct\_bfly32o(row1,row6, x1,x6,bias,shift); \

dct\_bfly32o(row2,row5, x2,x5,bias,shift); \

dct\_bfly32o(row3,row4, x3,x4,bias,shift); \

}

\_\_m128i rot0\_0 = dct\_const(stbi\_\_f2f(0.5411961f), stbi\_\_f2f(0.5411961f) + stbi\_\_f2f(-1.847759065f));

\_\_m128i rot0\_1 = dct\_const(stbi\_\_f2f(0.5411961f) + stbi\_\_f2f( 0.765366865f), stbi\_\_f2f(0.5411961f));

\_\_m128i rot1\_0 = dct\_const(stbi\_\_f2f(1.175875602f) + stbi\_\_f2f(-0.899976223f), stbi\_\_f2f(1.175875602f));

\_\_m128i rot1\_1 = dct\_const(stbi\_\_f2f(1.175875602f), stbi\_\_f2f(1.175875602f) + stbi\_\_f2f(-2.562915447f));

\_\_m128i rot2\_0 = dct\_const(stbi\_\_f2f(-1.961570560f) + stbi\_\_f2f( 0.298631336f), stbi\_\_f2f(-1.961570560f));

\_\_m128i rot2\_1 = dct\_const(stbi\_\_f2f(-1.961570560f), stbi\_\_f2f(-1.961570560f) + stbi\_\_f2f( 3.072711026f));

\_\_m128i rot3\_0 = dct\_const(stbi\_\_f2f(-0.390180644f) + stbi\_\_f2f( 2.053119869f), stbi\_\_f2f(-0.390180644f));

\_\_m128i rot3\_1 = dct\_const(stbi\_\_f2f(-0.390180644f), stbi\_\_f2f(-0.390180644f) + stbi\_\_f2f( 1.501321110f));

// rounding biases in column/row passes, see stbi\_\_idct\_block for explanation.

\_\_m128i bias\_0 = \_mm\_set1\_epi32(512);

\_\_m128i bias\_1 = \_mm\_set1\_epi32(65536 + (128<<17));

// load

row0 = \_mm\_load\_si128((const \_\_m128i \*) (data + 0\*8));

row1 = \_mm\_load\_si128((const \_\_m128i \*) (data + 1\*8));

row2 = \_mm\_load\_si128((const \_\_m128i \*) (data + 2\*8));

row3 = \_mm\_load\_si128((const \_\_m128i \*) (data + 3\*8));

row4 = \_mm\_load\_si128((const \_\_m128i \*) (data + 4\*8));

row5 = \_mm\_load\_si128((const \_\_m128i \*) (data + 5\*8));

row6 = \_mm\_load\_si128((const \_\_m128i \*) (data + 6\*8));

row7 = \_mm\_load\_si128((const \_\_m128i \*) (data + 7\*8));

// column pass

dct\_pass(bias\_0, 10);

{

// 16bit 8x8 transpose pass 1

dct\_interleave16(row0, row4);

dct\_interleave16(row1, row5);

dct\_interleave16(row2, row6);

dct\_interleave16(row3, row7);

// transpose pass 2

dct\_interleave16(row0, row2);

dct\_interleave16(row1, row3);

dct\_interleave16(row4, row6);

dct\_interleave16(row5, row7);

// transpose pass 3

dct\_interleave16(row0, row1);

dct\_interleave16(row2, row3);

dct\_interleave16(row4, row5);

dct\_interleave16(row6, row7);

}

// row pass

dct\_pass(bias\_1, 17);

{

// pack

\_\_m128i p0 = \_mm\_packus\_epi16(row0, row1); // a0a1a2a3...a7b0b1b2b3...b7

\_\_m128i p1 = \_mm\_packus\_epi16(row2, row3);

\_\_m128i p2 = \_mm\_packus\_epi16(row4, row5);

\_\_m128i p3 = \_mm\_packus\_epi16(row6, row7);

// 8bit 8x8 transpose pass 1

dct\_interleave8(p0, p2); // a0e0a1e1...

dct\_interleave8(p1, p3); // c0g0c1g1...

// transpose pass 2

dct\_interleave8(p0, p1); // a0c0e0g0...

dct\_interleave8(p2, p3); // b0d0f0h0...

// transpose pass 3

dct\_interleave8(p0, p2); // a0b0c0d0...

dct\_interleave8(p1, p3); // a4b4c4d4...

// store

\_mm\_storel\_epi64((\_\_m128i \*) out, p0); out += out\_stride;

\_mm\_storel\_epi64((\_\_m128i \*) out, \_mm\_shuffle\_epi32(p0, 0x4e)); out += out\_stride;

\_mm\_storel\_epi64((\_\_m128i \*) out, p2); out += out\_stride;

\_mm\_storel\_epi64((\_\_m128i \*) out, \_mm\_shuffle\_epi32(p2, 0x4e)); out += out\_stride;

\_mm\_storel\_epi64((\_\_m128i \*) out, p1); out += out\_stride;

\_mm\_storel\_epi64((\_\_m128i \*) out, \_mm\_shuffle\_epi32(p1, 0x4e)); out += out\_stride;

\_mm\_storel\_epi64((\_\_m128i \*) out, p3); out += out\_stride;

\_mm\_storel\_epi64((\_\_m128i \*) out, \_mm\_shuffle\_epi32(p3, 0x4e));

}

#undef dct\_const

#undef dct\_rot

#undef dct\_widen

#undef dct\_wadd

#undef dct\_wsub

#undef dct\_bfly32o

#undef dct\_interleave8

#undef dct\_interleave16

#undef dct\_pass

}

#endif // STBI\_SSE2

#ifdef STBI\_NEON

// NEON integer IDCT. should produce bit-identical

// results to the generic C version.

static void stbi\_\_idct\_simd(stbi\_uc \*out, int out\_stride, short data[64])

{

int16x8\_t row0, row1, row2, row3, row4, row5, row6, row7;

int16x4\_t rot0\_0 = vdup\_n\_s16(stbi\_\_f2f(0.5411961f));

int16x4\_t rot0\_1 = vdup\_n\_s16(stbi\_\_f2f(-1.847759065f));

int16x4\_t rot0\_2 = vdup\_n\_s16(stbi\_\_f2f( 0.765366865f));

int16x4\_t rot1\_0 = vdup\_n\_s16(stbi\_\_f2f( 1.175875602f));

int16x4\_t rot1\_1 = vdup\_n\_s16(stbi\_\_f2f(-0.899976223f));

int16x4\_t rot1\_2 = vdup\_n\_s16(stbi\_\_f2f(-2.562915447f));

int16x4\_t rot2\_0 = vdup\_n\_s16(stbi\_\_f2f(-1.961570560f));

int16x4\_t rot2\_1 = vdup\_n\_s16(stbi\_\_f2f(-0.390180644f));

int16x4\_t rot3\_0 = vdup\_n\_s16(stbi\_\_f2f( 0.298631336f));

int16x4\_t rot3\_1 = vdup\_n\_s16(stbi\_\_f2f( 2.053119869f));

int16x4\_t rot3\_2 = vdup\_n\_s16(stbi\_\_f2f( 3.072711026f));

int16x4\_t rot3\_3 = vdup\_n\_s16(stbi\_\_f2f( 1.501321110f));

#define dct\_long\_mul(out, inq, coeff) \

int32x4\_t out##\_l = vmull\_s16(vget\_low\_s16(inq), coeff); \

int32x4\_t out##\_h = vmull\_s16(vget\_high\_s16(inq), coeff)

#define dct\_long\_mac(out, acc, inq, coeff) \

int32x4\_t out##\_l = vmlal\_s16(acc##\_l, vget\_low\_s16(inq), coeff); \

int32x4\_t out##\_h = vmlal\_s16(acc##\_h, vget\_high\_s16(inq), coeff)

#define dct\_widen(out, inq) \

int32x4\_t out##\_l = vshll\_n\_s16(vget\_low\_s16(inq), 12); \

int32x4\_t out##\_h = vshll\_n\_s16(vget\_high\_s16(inq), 12)

// wide add

#define dct\_wadd(out, a, b) \

int32x4\_t out##\_l = vaddq\_s32(a##\_l, b##\_l); \

int32x4\_t out##\_h = vaddq\_s32(a##\_h, b##\_h)

// wide sub

#define dct\_wsub(out, a, b) \

int32x4\_t out##\_l = vsubq\_s32(a##\_l, b##\_l); \

int32x4\_t out##\_h = vsubq\_s32(a##\_h, b##\_h)

// butterfly a/b, then shift using "shiftop" by "s" and pack

#define dct\_bfly32o(out0,out1, a,b,shiftop,s) \

{ \

dct\_wadd(sum, a, b); \

dct\_wsub(dif, a, b); \

out0 = vcombine\_s16(shiftop(sum\_l, s), shiftop(sum\_h, s)); \

out1 = vcombine\_s16(shiftop(dif\_l, s), shiftop(dif\_h, s)); \

}

#define dct\_pass(shiftop, shift) \

{ \

/\* even part \*/ \

int16x8\_t sum26 = vaddq\_s16(row2, row6); \

dct\_long\_mul(p1e, sum26, rot0\_0); \

dct\_long\_mac(t2e, p1e, row6, rot0\_1); \

dct\_long\_mac(t3e, p1e, row2, rot0\_2); \

int16x8\_t sum04 = vaddq\_s16(row0, row4); \

int16x8\_t dif04 = vsubq\_s16(row0, row4); \

dct\_widen(t0e, sum04); \

dct\_widen(t1e, dif04); \

dct\_wadd(x0, t0e, t3e); \

dct\_wsub(x3, t0e, t3e); \

dct\_wadd(x1, t1e, t2e); \

dct\_wsub(x2, t1e, t2e); \

/\* odd part \*/ \

int16x8\_t sum15 = vaddq\_s16(row1, row5); \

int16x8\_t sum17 = vaddq\_s16(row1, row7); \

int16x8\_t sum35 = vaddq\_s16(row3, row5); \

int16x8\_t sum37 = vaddq\_s16(row3, row7); \

int16x8\_t sumodd = vaddq\_s16(sum17, sum35); \

dct\_long\_mul(p5o, sumodd, rot1\_0); \

dct\_long\_mac(p1o, p5o, sum17, rot1\_1); \

dct\_long\_mac(p2o, p5o, sum35, rot1\_2); \

dct\_long\_mul(p3o, sum37, rot2\_0); \

dct\_long\_mul(p4o, sum15, rot2\_1); \

dct\_wadd(sump13o, p1o, p3o); \

dct\_wadd(sump24o, p2o, p4o); \

dct\_wadd(sump23o, p2o, p3o); \

dct\_wadd(sump14o, p1o, p4o); \

dct\_long\_mac(x4, sump13o, row7, rot3\_0); \

dct\_long\_mac(x5, sump24o, row5, rot3\_1); \

dct\_long\_mac(x6, sump23o, row3, rot3\_2); \

dct\_long\_mac(x7, sump14o, row1, rot3\_3); \

dct\_bfly32o(row0,row7, x0,x7,shiftop,shift); \

dct\_bfly32o(row1,row6, x1,x6,shiftop,shift); \

dct\_bfly32o(row2,row5, x2,x5,shiftop,shift); \

dct\_bfly32o(row3,row4, x3,x4,shiftop,shift); \

}

// load

row0 = vld1q\_s16(data + 0\*8);

row1 = vld1q\_s16(data + 1\*8);

row2 = vld1q\_s16(data + 2\*8);

row3 = vld1q\_s16(data + 3\*8);

row4 = vld1q\_s16(data + 4\*8);

row5 = vld1q\_s16(data + 5\*8);

row6 = vld1q\_s16(data + 6\*8);

row7 = vld1q\_s16(data + 7\*8);

// add DC bias

row0 = vaddq\_s16(row0, vsetq\_lane\_s16(1024, vdupq\_n\_s16(0), 0));

// column pass

dct\_pass(vrshrn\_n\_s32, 10);

// 16bit 8x8 transpose

{

// these three map to a single VTRN.16, VTRN.32, and VSWP, respectively.

// whether compilers actually get this is another story, sadly.

#define dct\_trn16(x, y) { int16x8x2\_t t = vtrnq\_s16(x, y); x = t.val[0]; y = t.val[1]; }

#define dct\_trn32(x, y) { int32x4x2\_t t = vtrnq\_s32(vreinterpretq\_s32\_s16(x), vreinterpretq\_s32\_s16(y)); x = vreinterpretq\_s16\_s32(t.val[0]); y = vreinterpretq\_s16\_s32(t.val[1]); }

#define dct\_trn64(x, y) { int16x8\_t x0 = x; int16x8\_t y0 = y; x = vcombine\_s16(vget\_low\_s16(x0), vget\_low\_s16(y0)); y = vcombine\_s16(vget\_high\_s16(x0), vget\_high\_s16(y0)); }

// pass 1

dct\_trn16(row0, row1); // a0b0a2b2a4b4a6b6

dct\_trn16(row2, row3);

dct\_trn16(row4, row5);

dct\_trn16(row6, row7);

// pass 2

dct\_trn32(row0, row2); // a0b0c0d0a4b4c4d4

dct\_trn32(row1, row3);

dct\_trn32(row4, row6);

dct\_trn32(row5, row7);

// pass 3

dct\_trn64(row0, row4); // a0b0c0d0e0f0g0h0

dct\_trn64(row1, row5);

dct\_trn64(row2, row6);

dct\_trn64(row3, row7);

#undef dct\_trn16

#undef dct\_trn32

#undef dct\_trn64

}

// row pass

// vrshrn\_n\_s32 only supports shifts up to 16, we need

// 17. so do a non-rounding shift of 16 first then follow

// up with a rounding shift by 1.

dct\_pass(vshrn\_n\_s32, 16);

{

// pack and round

uint8x8\_t p0 = vqrshrun\_n\_s16(row0, 1);

uint8x8\_t p1 = vqrshrun\_n\_s16(row1, 1);

uint8x8\_t p2 = vqrshrun\_n\_s16(row2, 1);

uint8x8\_t p3 = vqrshrun\_n\_s16(row3, 1);

uint8x8\_t p4 = vqrshrun\_n\_s16(row4, 1);

uint8x8\_t p5 = vqrshrun\_n\_s16(row5, 1);

uint8x8\_t p6 = vqrshrun\_n\_s16(row6, 1);

uint8x8\_t p7 = vqrshrun\_n\_s16(row7, 1);

// again, these can translate into one instruction, but often don't.

#define dct\_trn8\_8(x, y) { uint8x8x2\_t t = vtrn\_u8(x, y); x = t.val[0]; y = t.val[1]; }

#define dct\_trn8\_16(x, y) { uint16x4x2\_t t = vtrn\_u16(vreinterpret\_u16\_u8(x), vreinterpret\_u16\_u8(y)); x = vreinterpret\_u8\_u16(t.val[0]); y = vreinterpret\_u8\_u16(t.val[1]); }

#define dct\_trn8\_32(x, y) { uint32x2x2\_t t = vtrn\_u32(vreinterpret\_u32\_u8(x), vreinterpret\_u32\_u8(y)); x = vreinterpret\_u8\_u32(t.val[0]); y = vreinterpret\_u8\_u32(t.val[1]); }

// sadly can't use interleaved stores here since we only write

// 8 bytes to each scan line!

// 8x8 8-bit transpose pass 1

dct\_trn8\_8(p0, p1);

dct\_trn8\_8(p2, p3);

dct\_trn8\_8(p4, p5);

dct\_trn8\_8(p6, p7);

// pass 2

dct\_trn8\_16(p0, p2);

dct\_trn8\_16(p1, p3);

dct\_trn8\_16(p4, p6);

dct\_trn8\_16(p5, p7);

// pass 3

dct\_trn8\_32(p0, p4);

dct\_trn8\_32(p1, p5);

dct\_trn8\_32(p2, p6);

dct\_trn8\_32(p3, p7);

// store

vst1\_u8(out, p0); out += out\_stride;

vst1\_u8(out, p1); out += out\_stride;

vst1\_u8(out, p2); out += out\_stride;

vst1\_u8(out, p3); out += out\_stride;

vst1\_u8(out, p4); out += out\_stride;

vst1\_u8(out, p5); out += out\_stride;

vst1\_u8(out, p6); out += out\_stride;

vst1\_u8(out, p7);

#undef dct\_trn8\_8

#undef dct\_trn8\_16

#undef dct\_trn8\_32

}

#undef dct\_long\_mul

#undef dct\_long\_mac

#undef dct\_widen

#undef dct\_wadd

#undef dct\_wsub

#undef dct\_bfly32o

#undef dct\_pass

}

#endif // STBI\_NEON

#define STBI\_\_MARKER\_none 0xff

// if there's a pending marker from the entropy stream, return that

// otherwise, fetch from the stream and get a marker. if there's no

// marker, return 0xff, which is never a valid marker value

static stbi\_uc stbi\_\_get\_marker(stbi\_\_jpeg \*j)

{

stbi\_uc x;

if (j->marker != STBI\_\_MARKER\_none) { x = j->marker; j->marker = STBI\_\_MARKER\_none; return x; }

x = stbi\_\_get8(j->s);

if (x != 0xff) return STBI\_\_MARKER\_none;

while (x == 0xff)

x = stbi\_\_get8(j->s); // consume repeated 0xff fill bytes

return x;

}

// in each scan, we'll have scan\_n components, and the order

// of the components is specified by order[]

#define STBI\_\_RESTART(x) ((x) >= 0xd0 && (x) <= 0xd7)

// after a restart interval, stbi\_\_jpeg\_reset the entropy decoder and

// the dc prediction

static void stbi\_\_jpeg\_reset(stbi\_\_jpeg \*j)

{

j->code\_bits = 0;

j->code\_buffer = 0;

j->nomore = 0;

j->img\_comp[0].dc\_pred = j->img\_comp[1].dc\_pred = j->img\_comp[2].dc\_pred = j->img\_comp[3].dc\_pred = 0;

j->marker = STBI\_\_MARKER\_none;

j->todo = j->restart\_interval ? j->restart\_interval : 0x7fffffff;

j->eob\_run = 0;

// no more than 1<<31 MCUs if no restart\_interal? that's plenty safe,

// since we don't even allow 1<<30 pixels

}

static int stbi\_\_parse\_entropy\_coded\_data(stbi\_\_jpeg \*z)

{

stbi\_\_jpeg\_reset(z);

if (!z->progressive) {

if (z->scan\_n == 1) {

int i,j;

STBI\_SIMD\_ALIGN(short, data[64]);

int n = z->order[0];

// non-interleaved data, we just need to process one block at a time,

// in trivial scanline order

// number of blocks to do just depends on how many actual "pixels" this

// component has, independent of interleaved MCU blocking and such

int w = (z->img\_comp[n].x+7) >> 3;

int h = (z->img\_comp[n].y+7) >> 3;

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

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

int ha = z->img\_comp[n].ha;

if (!stbi\_\_jpeg\_decode\_block(z, data, z->huff\_dc+z->img\_comp[n].hd, z->huff\_ac+ha, z->fast\_ac[ha], n, z->dequant[z->img\_comp[n].tq])) return 0;

z->idct\_block\_kernel(z->img\_comp[n].data+z->img\_comp[n].w2\*j\*8+i\*8, z->img\_comp[n].w2, data);

// every data block is an MCU, so countdown the restart interval

if (--z->todo <= 0) {

if (z->code\_bits < 24) stbi\_\_grow\_buffer\_unsafe(z);

// if it's NOT a restart, then just bail, so we get corrupt data

// rather than no data

if (!STBI\_\_RESTART(z->marker)) return 1;

stbi\_\_jpeg\_reset(z);

}

}

}

return 1;

} else { // interleaved

int i,j,k,x,y;

STBI\_SIMD\_ALIGN(short, data[64]);

for (j=0; j < z->img\_mcu\_y; ++j) {

for (i=0; i < z->img\_mcu\_x; ++i) {

// scan an interleaved mcu... process scan\_n components in order

for (k=0; k < z->scan\_n; ++k) {

int n = z->order[k];

// scan out an mcu's worth of this component; that's just determined

// by the basic H and V specified for the component

for (y=0; y < z->img\_comp[n].v; ++y) {

for (x=0; x < z->img\_comp[n].h; ++x) {

int x2 = (i\*z->img\_comp[n].h + x)\*8;

int y2 = (j\*z->img\_comp[n].v + y)\*8;

int ha = z->img\_comp[n].ha;

if (!stbi\_\_jpeg\_decode\_block(z, data, z->huff\_dc+z->img\_comp[n].hd, z->huff\_ac+ha, z->fast\_ac[ha], n, z->dequant[z->img\_comp[n].tq])) return 0;

z->idct\_block\_kernel(z->img\_comp[n].data+z->img\_comp[n].w2\*y2+x2, z->img\_comp[n].w2, data);

}

}

}

// after all interleaved components, that's an interleaved MCU,

// so now count down the restart interval

if (--z->todo <= 0) {

if (z->code\_bits < 24) stbi\_\_grow\_buffer\_unsafe(z);

if (!STBI\_\_RESTART(z->marker)) return 1;

stbi\_\_jpeg\_reset(z);

}

}

}

return 1;

}

} else {

if (z->scan\_n == 1) {

int i,j;

int n = z->order[0];

// non-interleaved data, we just need to process one block at a time,

// in trivial scanline order

// number of blocks to do just depends on how many actual "pixels" this

// component has, independent of interleaved MCU blocking and such

int w = (z->img\_comp[n].x+7) >> 3;

int h = (z->img\_comp[n].y+7) >> 3;

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

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

short \*data = z->img\_comp[n].coeff + 64 \* (i + j \* z->img\_comp[n].coeff\_w);

if (z->spec\_start == 0) {

if (!stbi\_\_jpeg\_decode\_block\_prog\_dc(z, data, &z->huff\_dc[z->img\_comp[n].hd], n))

return 0;

} else {

int ha = z->img\_comp[n].ha;

if (!stbi\_\_jpeg\_decode\_block\_prog\_ac(z, data, &z->huff\_ac[ha], z->fast\_ac[ha]))

return 0;

}

// every data block is an MCU, so countdown the restart interval

if (--z->todo <= 0) {

if (z->code\_bits < 24) stbi\_\_grow\_buffer\_unsafe(z);

if (!STBI\_\_RESTART(z->marker)) return 1;

stbi\_\_jpeg\_reset(z);

}

}

}

return 1;

} else { // interleaved

int i,j,k,x,y;

for (j=0; j < z->img\_mcu\_y; ++j) {

for (i=0; i < z->img\_mcu\_x; ++i) {

// scan an interleaved mcu... process scan\_n components in order

for (k=0; k < z->scan\_n; ++k) {

int n = z->order[k];

// scan out an mcu's worth of this component; that's just determined

// by the basic H and V specified for the component

for (y=0; y < z->img\_comp[n].v; ++y) {

for (x=0; x < z->img\_comp[n].h; ++x) {

int x2 = (i\*z->img\_comp[n].h + x);

int y2 = (j\*z->img\_comp[n].v + y);

short \*data = z->img\_comp[n].coeff + 64 \* (x2 + y2 \* z->img\_comp[n].coeff\_w);

if (!stbi\_\_jpeg\_decode\_block\_prog\_dc(z, data, &z->huff\_dc[z->img\_comp[n].hd], n))

return 0;

}

}

}

// after all interleaved components, that's an interleaved MCU,

// so now count down the restart interval

if (--z->todo <= 0) {

if (z->code\_bits < 24) stbi\_\_grow\_buffer\_unsafe(z);

if (!STBI\_\_RESTART(z->marker)) return 1;

stbi\_\_jpeg\_reset(z);

}

}

}

return 1;

}

}

}

static void stbi\_\_jpeg\_dequantize(short \*data, stbi\_\_uint16 \*dequant)

{

int i;

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

data[i] \*= dequant[i];

}

static void stbi\_\_jpeg\_finish(stbi\_\_jpeg \*z)

{

if (z->progressive) {

// dequantize and idct the data

int i,j,n;

for (n=0; n < z->s->img\_n; ++n) {

int w = (z->img\_comp[n].x+7) >> 3;

int h = (z->img\_comp[n].y+7) >> 3;

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

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

short \*data = z->img\_comp[n].coeff + 64 \* (i + j \* z->img\_comp[n].coeff\_w);

stbi\_\_jpeg\_dequantize(data, z->dequant[z->img\_comp[n].tq]);

z->idct\_block\_kernel(z->img\_comp[n].data+z->img\_comp[n].w2\*j\*8+i\*8, z->img\_comp[n].w2, data);

}

}

}

}

}

static int stbi\_\_process\_marker(stbi\_\_jpeg \*z, int m)

{

int L;

switch (m) {

case STBI\_\_MARKER\_none: // no marker found

return stbi\_\_err("expected marker","Corrupt JPEG");

case 0xDD: // DRI - specify restart interval

if (stbi\_\_get16be(z->s) != 4) return stbi\_\_err("bad DRI len","Corrupt JPEG");

z->restart\_interval = stbi\_\_get16be(z->s);

return 1;

case 0xDB: // DQT - define quantization table

L = stbi\_\_get16be(z->s)-2;

while (L > 0) {

int q = stbi\_\_get8(z->s);

int p = q >> 4, sixteen = (p != 0);

int t = q & 15,i;

if (p != 0 && p != 1) return stbi\_\_err("bad DQT type","Corrupt JPEG");

if (t > 3) return stbi\_\_err("bad DQT table","Corrupt JPEG");

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

z->dequant[t][stbi\_\_jpeg\_dezigzag[i]] = (stbi\_\_uint16)(sixteen ? stbi\_\_get16be(z->s) : stbi\_\_get8(z->s));

L -= (sixteen ? 129 : 65);

}

return L==0;

case 0xC4: // DHT - define huffman table

L = stbi\_\_get16be(z->s)-2;

while (L > 0) {

stbi\_uc \*v;

int sizes[16],i,n=0;

int q = stbi\_\_get8(z->s);

int tc = q >> 4;

int th = q & 15;

if (tc > 1 || th > 3) return stbi\_\_err("bad DHT header","Corrupt JPEG");

for (i=0; i < 16; ++i) {

sizes[i] = stbi\_\_get8(z->s);

n += sizes[i];

}

L -= 17;

if (tc == 0) {

if (!stbi\_\_build\_huffman(z->huff\_dc+th, sizes)) return 0;

v = z->huff\_dc[th].values;

} else {

if (!stbi\_\_build\_huffman(z->huff\_ac+th, sizes)) return 0;

v = z->huff\_ac[th].values;

}

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

v[i] = stbi\_\_get8(z->s);

if (tc != 0)

stbi\_\_build\_fast\_ac(z->fast\_ac[th], z->huff\_ac + th);

L -= n;

}

return L==0;

}

// check for comment block or APP blocks

if ((m >= 0xE0 && m <= 0xEF) || m == 0xFE) {

L = stbi\_\_get16be(z->s);

if (L < 2) {

if (m == 0xFE)

return stbi\_\_err("bad COM len","Corrupt JPEG");

else

return stbi\_\_err("bad APP len","Corrupt JPEG");

}

L -= 2;

if (m == 0xE0 && L >= 5) { // JFIF APP0 segment

static const unsigned char tag[5] = {'J','F','I','F','\0'};

int ok = 1;

int i;

for (i=0; i < 5; ++i)

if (stbi\_\_get8(z->s) != tag[i])

ok = 0;

L -= 5;

if (ok)

z->jfif = 1;

} else if (m == 0xEE && L >= 12) { // Adobe APP14 segment

static const unsigned char tag[6] = {'A','d','o','b','e','\0'};

int ok = 1;

int i;

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

if (stbi\_\_get8(z->s) != tag[i])

ok = 0;

L -= 6;

if (ok) {

stbi\_\_get8(z->s); // version

stbi\_\_get16be(z->s); // flags0

stbi\_\_get16be(z->s); // flags1

z->app14\_color\_transform = stbi\_\_get8(z->s); // color transform

L -= 6;

}

}

stbi\_\_skip(z->s, L);

return 1;

}

return stbi\_\_err("unknown marker","Corrupt JPEG");

}

// after we see SOS

static int stbi\_\_process\_scan\_header(stbi\_\_jpeg \*z)

{

int i;

int Ls = stbi\_\_get16be(z->s);

z->scan\_n = stbi\_\_get8(z->s);

if (z->scan\_n < 1 || z->scan\_n > 4 || z->scan\_n > (int) z->s->img\_n) return stbi\_\_err("bad SOS component count","Corrupt JPEG");

if (Ls != 6+2\*z->scan\_n) return stbi\_\_err("bad SOS len","Corrupt JPEG");

for (i=0; i < z->scan\_n; ++i) {

int id = stbi\_\_get8(z->s), which;

int q = stbi\_\_get8(z->s);

for (which = 0; which < z->s->img\_n; ++which)

if (z->img\_comp[which].id == id)

break;

if (which == z->s->img\_n) return 0; // no match

z->img\_comp[which].hd = q >> 4; if (z->img\_comp[which].hd > 3) return stbi\_\_err("bad DC huff","Corrupt JPEG");

z->img\_comp[which].ha = q & 15; if (z->img\_comp[which].ha > 3) return stbi\_\_err("bad AC huff","Corrupt JPEG");

z->order[i] = which;

}

{

int aa;

z->spec\_start = stbi\_\_get8(z->s);

z->spec\_end = stbi\_\_get8(z->s); // should be 63, but might be 0

aa = stbi\_\_get8(z->s);

z->succ\_high = (aa >> 4);

z->succ\_low = (aa & 15);

if (z->progressive) {

if (z->spec\_start > 63 || z->spec\_end > 63 || z->spec\_start > z->spec\_end || z->succ\_high > 13 || z->succ\_low > 13)

return stbi\_\_err("bad SOS", "Corrupt JPEG");

} else {

if (z->spec\_start != 0) return stbi\_\_err("bad SOS","Corrupt JPEG");

if (z->succ\_high != 0 || z->succ\_low != 0) return stbi\_\_err("bad SOS","Corrupt JPEG");

z->spec\_end = 63;

}

}

return 1;

}

static int stbi\_\_free\_jpeg\_components(stbi\_\_jpeg \*z, int ncomp, int why)

{

int i;

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

if (z->img\_comp[i].raw\_data) {

STBI\_FREE(z->img\_comp[i].raw\_data);

z->img\_comp[i].raw\_data = NULL;

z->img\_comp[i].data = NULL;

}

if (z->img\_comp[i].raw\_coeff) {

STBI\_FREE(z->img\_comp[i].raw\_coeff);

z->img\_comp[i].raw\_coeff = 0;

z->img\_comp[i].coeff = 0;

}

if (z->img\_comp[i].linebuf) {

STBI\_FREE(z->img\_comp[i].linebuf);

z->img\_comp[i].linebuf = NULL;

}

}

return why;

}

static int stbi\_\_process\_frame\_header(stbi\_\_jpeg \*z, int scan)

{

stbi\_\_context \*s = z->s;

int Lf,p,i,q, h\_max=1,v\_max=1,c;

Lf = stbi\_\_get16be(s); if (Lf < 11) return stbi\_\_err("bad SOF len","Corrupt JPEG"); // JPEG

p = stbi\_\_get8(s); if (p != 8) return stbi\_\_err("only 8-bit","JPEG format not supported: 8-bit only"); // JPEG baseline

s->img\_y = stbi\_\_get16be(s); if (s->img\_y == 0) return stbi\_\_err("no header height", "JPEG format not supported: delayed height"); // Legal, but we don't handle it--but neither does IJG

s->img\_x = stbi\_\_get16be(s); if (s->img\_x == 0) return stbi\_\_err("0 width","Corrupt JPEG"); // JPEG requires

c = stbi\_\_get8(s);

if (c != 3 && c != 1 && c != 4) return stbi\_\_err("bad component count","Corrupt JPEG");

s->img\_n = c;

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

z->img\_comp[i].data = NULL;

z->img\_comp[i].linebuf = NULL;

}

if (Lf != 8+3\*s->img\_n) return stbi\_\_err("bad SOF len","Corrupt JPEG");

z->rgb = 0;

for (i=0; i < s->img\_n; ++i) {

static unsigned char rgb[3] = { 'R', 'G', 'B' };

z->img\_comp[i].id = stbi\_\_get8(s);

if (s->img\_n == 3 && z->img\_comp[i].id == rgb[i])

++z->rgb;

q = stbi\_\_get8(s);

z->img\_comp[i].h = (q >> 4); if (!z->img\_comp[i].h || z->img\_comp[i].h > 4) return stbi\_\_err("bad H","Corrupt JPEG");

z->img\_comp[i].v = q & 15; if (!z->img\_comp[i].v || z->img\_comp[i].v > 4) return stbi\_\_err("bad V","Corrupt JPEG");

z->img\_comp[i].tq = stbi\_\_get8(s); if (z->img\_comp[i].tq > 3) return stbi\_\_err("bad TQ","Corrupt JPEG");

}

if (scan != STBI\_\_SCAN\_load) return 1;

if (!stbi\_\_mad3sizes\_valid(s->img\_x, s->img\_y, s->img\_n, 0)) return stbi\_\_err("too large", "Image too large to decode");

for (i=0; i < s->img\_n; ++i) {

if (z->img\_comp[i].h > h\_max) h\_max = z->img\_comp[i].h;

if (z->img\_comp[i].v > v\_max) v\_max = z->img\_comp[i].v;

}

// compute interleaved mcu info

z->img\_h\_max = h\_max;

z->img\_v\_max = v\_max;

z->img\_mcu\_w = h\_max \* 8;

z->img\_mcu\_h = v\_max \* 8;

// these sizes can't be more than 17 bits

z->img\_mcu\_x = (s->img\_x + z->img\_mcu\_w-1) / z->img\_mcu\_w;

z->img\_mcu\_y = (s->img\_y + z->img\_mcu\_h-1) / z->img\_mcu\_h;

for (i=0; i < s->img\_n; ++i) {

// number of effective pixels (e.g. for non-interleaved MCU)

z->img\_comp[i].x = (s->img\_x \* z->img\_comp[i].h + h\_max-1) / h\_max;

z->img\_comp[i].y = (s->img\_y \* z->img\_comp[i].v + v\_max-1) / v\_max;

// to simplify generation, we'll allocate enough memory to decode

// the bogus oversized data from using interleaved MCUs and their

// big blocks (e.g. a 16x16 iMCU on an image of width 33); we won't

// discard the extra data until colorspace conversion

//

// img\_mcu\_x, img\_mcu\_y: <=17 bits; comp[i].h and .v are <=4 (checked earlier)

// so these muls can't overflow with 32-bit ints (which we require)

z->img\_comp[i].w2 = z->img\_mcu\_x \* z->img\_comp[i].h \* 8;

z->img\_comp[i].h2 = z->img\_mcu\_y \* z->img\_comp[i].v \* 8;

z->img\_comp[i].coeff = 0;

z->img\_comp[i].raw\_coeff = 0;

z->img\_comp[i].linebuf = NULL;

z->img\_comp[i].raw\_data = stbi\_\_malloc\_mad2(z->img\_comp[i].w2, z->img\_comp[i].h2, 15);

if (z->img\_comp[i].raw\_data == NULL)

return stbi\_\_free\_jpeg\_components(z, i+1, stbi\_\_err("outofmem", "Out of memory"));

// align blocks for idct using mmx/sse

z->img\_comp[i].data = (stbi\_uc\*) (((size\_t) z->img\_comp[i].raw\_data + 15) & ~15);

if (z->progressive) {

// w2, h2 are multiples of 8 (see above)

z->img\_comp[i].coeff\_w = z->img\_comp[i].w2 / 8;

z->img\_comp[i].coeff\_h = z->img\_comp[i].h2 / 8;

z->img\_comp[i].raw\_coeff = stbi\_\_malloc\_mad3(z->img\_comp[i].w2, z->img\_comp[i].h2, sizeof(short), 15);

if (z->img\_comp[i].raw\_coeff == NULL)

return stbi\_\_free\_jpeg\_components(z, i+1, stbi\_\_err("outofmem", "Out of memory"));

z->img\_comp[i].coeff = (short\*) (((size\_t) z->img\_comp[i].raw\_coeff + 15) & ~15);

}

}

return 1;

}

// use comparisons since in some cases we handle more than one case (e.g. SOF)

#define stbi\_\_DNL(x) ((x) == 0xdc)

#define stbi\_\_SOI(x) ((x) == 0xd8)

#define stbi\_\_EOI(x) ((x) == 0xd9)

#define stbi\_\_SOF(x) ((x) == 0xc0 || (x) == 0xc1 || (x) == 0xc2)

#define stbi\_\_SOS(x) ((x) == 0xda)

#define stbi\_\_SOF\_progressive(x) ((x) == 0xc2)

static int stbi\_\_decode\_jpeg\_header(stbi\_\_jpeg \*z, int scan)

{

int m;

z->jfif = 0;

z->app14\_color\_transform = -1; // valid values are 0,1,2

z->marker = STBI\_\_MARKER\_none; // initialize cached marker to empty

m = stbi\_\_get\_marker(z);

if (!stbi\_\_SOI(m)) return stbi\_\_err("no SOI","Corrupt JPEG");

if (scan == STBI\_\_SCAN\_type) return 1;

m = stbi\_\_get\_marker(z);

while (!stbi\_\_SOF(m)) {

if (!stbi\_\_process\_marker(z,m)) return 0;

m = stbi\_\_get\_marker(z);

while (m == STBI\_\_MARKER\_none) {

// some files have extra padding after their blocks, so ok, we'll scan

if (stbi\_\_at\_eof(z->s)) return stbi\_\_err("no SOF", "Corrupt JPEG");

m = stbi\_\_get\_marker(z);

}

}

z->progressive = stbi\_\_SOF\_progressive(m);

if (!stbi\_\_process\_frame\_header(z, scan)) return 0;

return 1;

}

// decode image to YCbCr format

static int stbi\_\_decode\_jpeg\_image(stbi\_\_jpeg \*j)

{

int m;

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

j->img\_comp[m].raw\_data = NULL;

j->img\_comp[m].raw\_coeff = NULL;

}

j->restart\_interval = 0;

if (!stbi\_\_decode\_jpeg\_header(j, STBI\_\_SCAN\_load)) return 0;

m = stbi\_\_get\_marker(j);

while (!stbi\_\_EOI(m)) {

if (stbi\_\_SOS(m)) {

if (!stbi\_\_process\_scan\_header(j)) return 0;

if (!stbi\_\_parse\_entropy\_coded\_data(j)) return 0;

if (j->marker == STBI\_\_MARKER\_none ) {

// handle 0s at the end of image data from IP Kamera 9060

while (!stbi\_\_at\_eof(j->s)) {

int x = stbi\_\_get8(j->s);

if (x == 255) {

j->marker = stbi\_\_get8(j->s);

break;

}

}

// if we reach eof without hitting a marker, stbi\_\_get\_marker() below will fail and we'll eventually return 0

}

} else if (stbi\_\_DNL(m)) {

int Ld = stbi\_\_get16be(j->s);

stbi\_\_uint32 NL = stbi\_\_get16be(j->s);

if (Ld != 4) stbi\_\_err("bad DNL len", "Corrupt JPEG");

if (NL != j->s->img\_y) stbi\_\_err("bad DNL height", "Corrupt JPEG");

} else {

if (!stbi\_\_process\_marker(j, m)) return 0;

}

m = stbi\_\_get\_marker(j);

}

if (j->progressive)

stbi\_\_jpeg\_finish(j);

return 1;

}

// static jfif-centered resampling (across block boundaries)

typedef stbi\_uc \*(\*resample\_row\_func)(stbi\_uc \*out, stbi\_uc \*in0, stbi\_uc \*in1,

int w, int hs);

#define stbi\_\_div4(x) ((stbi\_uc) ((x) >> 2))

static stbi\_uc \*resample\_row\_1(stbi\_uc \*out, stbi\_uc \*in\_near, stbi\_uc \*in\_far, int w, int hs)

{

STBI\_NOTUSED(out);

STBI\_NOTUSED(in\_far);

STBI\_NOTUSED(w);

STBI\_NOTUSED(hs);

return in\_near;

}

static stbi\_uc\* stbi\_\_resample\_row\_v\_2(stbi\_uc \*out, stbi\_uc \*in\_near, stbi\_uc \*in\_far, int w, int hs)

{

// need to generate two samples vertically for every one in input

int i;

STBI\_NOTUSED(hs);

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

out[i] = stbi\_\_div4(3\*in\_near[i] + in\_far[i] + 2);

return out;

}

static stbi\_uc\* stbi\_\_resample\_row\_h\_2(stbi\_uc \*out, stbi\_uc \*in\_near, stbi\_uc \*in\_far, int w, int hs)

{

// need to generate two samples horizontally for every one in input

int i;

stbi\_uc \*input = in\_near;

if (w == 1) {

// if only one sample, can't do any interpolation

out[0] = out[1] = input[0];

return out;

}

out[0] = input[0];

out[1] = stbi\_\_div4(input[0]\*3 + input[1] + 2);

for (i=1; i < w-1; ++i) {

int n = 3\*input[i]+2;

out[i\*2+0] = stbi\_\_div4(n+input[i-1]);

out[i\*2+1] = stbi\_\_div4(n+input[i+1]);

}

out[i\*2+0] = stbi\_\_div4(input[w-2]\*3 + input[w-1] + 2);

out[i\*2+1] = input[w-1];

STBI\_NOTUSED(in\_far);

STBI\_NOTUSED(hs);

return out;

}

#define stbi\_\_div16(x) ((stbi\_uc) ((x) >> 4))

static stbi\_uc \*stbi\_\_resample\_row\_hv\_2(stbi\_uc \*out, stbi\_uc \*in\_near, stbi\_uc \*in\_far, int w, int hs)

{

// need to generate 2x2 samples for every one in input

int i,t0,t1;

if (w == 1) {

out[0] = out[1] = stbi\_\_div4(3\*in\_near[0] + in\_far[0] + 2);

return out;

}

t1 = 3\*in\_near[0] + in\_far[0];

out[0] = stbi\_\_div4(t1+2);

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

t0 = t1;

t1 = 3\*in\_near[i]+in\_far[i];

out[i\*2-1] = stbi\_\_div16(3\*t0 + t1 + 8);

out[i\*2 ] = stbi\_\_div16(3\*t1 + t0 + 8);

}

out[w\*2-1] = stbi\_\_div4(t1+2);

STBI\_NOTUSED(hs);

return out;

}

#if defined(STBI\_SSE2) || defined(STBI\_NEON)

static stbi\_uc \*stbi\_\_resample\_row\_hv\_2\_simd(stbi\_uc \*out, stbi\_uc \*in\_near, stbi\_uc \*in\_far, int w, int hs)

{

// need to generate 2x2 samples for every one in input

int i=0,t0,t1;

if (w == 1) {

out[0] = out[1] = stbi\_\_div4(3\*in\_near[0] + in\_far[0] + 2);

return out;

}

t1 = 3\*in\_near[0] + in\_far[0];

// process groups of 8 pixels for as long as we can.

// note we can't handle the last pixel in a row in this loop

// because we need to handle the filter boundary conditions.

for (; i < ((w-1) & ~7); i += 8) {

#if defined(STBI\_SSE2)

// load and perform the vertical filtering pass

// this uses 3\*x + y = 4\*x + (y - x)

\_\_m128i zero = \_mm\_setzero\_si128();

\_\_m128i farb = \_mm\_loadl\_epi64((\_\_m128i \*) (in\_far + i));

\_\_m128i nearb = \_mm\_loadl\_epi64((\_\_m128i \*) (in\_near + i));

\_\_m128i farw = \_mm\_unpacklo\_epi8(farb, zero);

\_\_m128i nearw = \_mm\_unpacklo\_epi8(nearb, zero);

\_\_m128i diff = \_mm\_sub\_epi16(farw, nearw);

\_\_m128i nears = \_mm\_slli\_epi16(nearw, 2);

\_\_m128i curr = \_mm\_add\_epi16(nears, diff); // current row

// horizontal filter works the same based on shifted vers of current

// row. "prev" is current row shifted right by 1 pixel; we need to

// insert the previous pixel value (from t1).

// "next" is current row shifted left by 1 pixel, with first pixel

// of next block of 8 pixels added in.

\_\_m128i prv0 = \_mm\_slli\_si128(curr, 2);

\_\_m128i nxt0 = \_mm\_srli\_si128(curr, 2);

\_\_m128i prev = \_mm\_insert\_epi16(prv0, t1, 0);

\_\_m128i next = \_mm\_insert\_epi16(nxt0, 3\*in\_near[i+8] + in\_far[i+8], 7);

// horizontal filter, polyphase implementation since it's convenient:

// even pixels = 3\*cur + prev = cur\*4 + (prev - cur)

// odd pixels = 3\*cur + next = cur\*4 + (next - cur)

// note the shared term.

\_\_m128i bias = \_mm\_set1\_epi16(8);

\_\_m128i curs = \_mm\_slli\_epi16(curr, 2);

\_\_m128i prvd = \_mm\_sub\_epi16(prev, curr);

\_\_m128i nxtd = \_mm\_sub\_epi16(next, curr);

\_\_m128i curb = \_mm\_add\_epi16(curs, bias);

\_\_m128i even = \_mm\_add\_epi16(prvd, curb);

\_\_m128i odd = \_mm\_add\_epi16(nxtd, curb);

// interleave even and odd pixels, then undo scaling.

\_\_m128i int0 = \_mm\_unpacklo\_epi16(even, odd);

\_\_m128i int1 = \_mm\_unpackhi\_epi16(even, odd);

\_\_m128i de0 = \_mm\_srli\_epi16(int0, 4);

\_\_m128i de1 = \_mm\_srli\_epi16(int1, 4);

// pack and write output

\_\_m128i outv = \_mm\_packus\_epi16(de0, de1);

\_mm\_storeu\_si128((\_\_m128i \*) (out + i\*2), outv);

#elif defined(STBI\_NEON)

// load and perform the vertical filtering pass

// this uses 3\*x + y = 4\*x + (y - x)

uint8x8\_t farb = vld1\_u8(in\_far + i);

uint8x8\_t nearb = vld1\_u8(in\_near + i);

int16x8\_t diff = vreinterpretq\_s16\_u16(vsubl\_u8(farb, nearb));

int16x8\_t nears = vreinterpretq\_s16\_u16(vshll\_n\_u8(nearb, 2));

int16x8\_t curr = vaddq\_s16(nears, diff); // current row

// horizontal filter works the same based on shifted vers of current

// row. "prev" is current row shifted right by 1 pixel; we need to

// insert the previous pixel value (from t1).

// "next" is current row shifted left by 1 pixel, with first pixel

// of next block of 8 pixels added in.

int16x8\_t prv0 = vextq\_s16(curr, curr, 7);

int16x8\_t nxt0 = vextq\_s16(curr, curr, 1);

int16x8\_t prev = vsetq\_lane\_s16(t1, prv0, 0);

int16x8\_t next = vsetq\_lane\_s16(3\*in\_near[i+8] + in\_far[i+8], nxt0, 7);

// horizontal filter, polyphase implementation since it's convenient:

// even pixels = 3\*cur + prev = cur\*4 + (prev - cur)

// odd pixels = 3\*cur + next = cur\*4 + (next - cur)

// note the shared term.

int16x8\_t curs = vshlq\_n\_s16(curr, 2);

int16x8\_t prvd = vsubq\_s16(prev, curr);

int16x8\_t nxtd = vsubq\_s16(next, curr);

int16x8\_t even = vaddq\_s16(curs, prvd);

int16x8\_t odd = vaddq\_s16(curs, nxtd);

// undo scaling and round, then store with even/odd phases interleaved

uint8x8x2\_t o;

o.val[0] = vqrshrun\_n\_s16(even, 4);

o.val[1] = vqrshrun\_n\_s16(odd, 4);

vst2\_u8(out + i\*2, o);

#endif

// "previous" value for next iter

t1 = 3\*in\_near[i+7] + in\_far[i+7];

}

t0 = t1;

t1 = 3\*in\_near[i] + in\_far[i];

out[i\*2] = stbi\_\_div16(3\*t1 + t0 + 8);

for (++i; i < w; ++i) {

t0 = t1;

t1 = 3\*in\_near[i]+in\_far[i];

out[i\*2-1] = stbi\_\_div16(3\*t0 + t1 + 8);

out[i\*2 ] = stbi\_\_div16(3\*t1 + t0 + 8);

}

out[w\*2-1] = stbi\_\_div4(t1+2);

STBI\_NOTUSED(hs);

return out;

}

#endif

static stbi\_uc \*stbi\_\_resample\_row\_generic(stbi\_uc \*out, stbi\_uc \*in\_near, stbi\_uc \*in\_far, int w, int hs)

{

// resample with nearest-neighbor

int i,j;

STBI\_NOTUSED(in\_far);

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

for (j=0; j < hs; ++j)

out[i\*hs+j] = in\_near[i];

return out;

}

// this is a reduced-precision calculation of YCbCr-to-RGB introduced

// to make sure the code produces the same results in both SIMD and scalar

#define stbi\_\_float2fixed(x) (((int) ((x) \* 4096.0f + 0.5f)) << 8)

static void stbi\_\_YCbCr\_to\_RGB\_row(stbi\_uc \*out, const stbi\_uc \*y, const stbi\_uc \*pcb, const stbi\_uc \*pcr, int count, int step)

{

int i;

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

int y\_fixed = (y[i] << 20) + (1<<19); // rounding

int r,g,b;

int cr = pcr[i] - 128;

int cb = pcb[i] - 128;

r = y\_fixed + cr\* stbi\_\_float2fixed(1.40200f);

g = y\_fixed + (cr\*-stbi\_\_float2fixed(0.71414f)) + ((cb\*-stbi\_\_float2fixed(0.34414f)) & 0xffff0000);

b = y\_fixed + cb\* stbi\_\_float2fixed(1.77200f);

r >>= 20;

g >>= 20;

b >>= 20;

if ((unsigned) r > 255) { if (r < 0) r = 0; else r = 255; }

if ((unsigned) g > 255) { if (g < 0) g = 0; else g = 255; }

if ((unsigned) b > 255) { if (b < 0) b = 0; else b = 255; }

out[0] = (stbi\_uc)r;

out[1] = (stbi\_uc)g;

out[2] = (stbi\_uc)b;

out[3] = 255;

out += step;

}

}

#if defined(STBI\_SSE2) || defined(STBI\_NEON)

static void stbi\_\_YCbCr\_to\_RGB\_simd(stbi\_uc \*out, stbi\_uc const \*y, stbi\_uc const \*pcb, stbi\_uc const \*pcr, int count, int step)

{

int i = 0;

#ifdef STBI\_SSE2

// step == 3 is pretty ugly on the final interleave, and i'm not convinced

// it's useful in practice (you wouldn't use it for textures, for example).

// so just accelerate step == 4 case.

if (step == 4) {

// this is a fairly straightforward implementation and not super-optimized.

\_\_m128i signflip = \_mm\_set1\_epi8(-0x80);

\_\_m128i cr\_const0 = \_mm\_set1\_epi16( (short) ( 1.40200f\*4096.0f+0.5f));

\_\_m128i cr\_const1 = \_mm\_set1\_epi16( - (short) ( 0.71414f\*4096.0f+0.5f));

\_\_m128i cb\_const0 = \_mm\_set1\_epi16( - (short) ( 0.34414f\*4096.0f+0.5f));

\_\_m128i cb\_const1 = \_mm\_set1\_epi16( (short) ( 1.77200f\*4096.0f+0.5f));

\_\_m128i y\_bias = \_mm\_set1\_epi8((char) (unsigned char) 128);

\_\_m128i xw = \_mm\_set1\_epi16(255); // alpha channel

for (; i+7 < count; i += 8) {

// load

\_\_m128i y\_bytes = \_mm\_loadl\_epi64((\_\_m128i \*) (y+i));

\_\_m128i cr\_bytes = \_mm\_loadl\_epi64((\_\_m128i \*) (pcr+i));

\_\_m128i cb\_bytes = \_mm\_loadl\_epi64((\_\_m128i \*) (pcb+i));

\_\_m128i cr\_biased = \_mm\_xor\_si128(cr\_bytes, signflip); // -128

\_\_m128i cb\_biased = \_mm\_xor\_si128(cb\_bytes, signflip); // -128

// unpack to short (and left-shift cr, cb by 8)

\_\_m128i yw = \_mm\_unpacklo\_epi8(y\_bias, y\_bytes);

\_\_m128i crw = \_mm\_unpacklo\_epi8(\_mm\_setzero\_si128(), cr\_biased);

\_\_m128i cbw = \_mm\_unpacklo\_epi8(\_mm\_setzero\_si128(), cb\_biased);

// color transform

\_\_m128i yws = \_mm\_srli\_epi16(yw, 4);

\_\_m128i cr0 = \_mm\_mulhi\_epi16(cr\_const0, crw);

\_\_m128i cb0 = \_mm\_mulhi\_epi16(cb\_const0, cbw);

\_\_m128i cb1 = \_mm\_mulhi\_epi16(cbw, cb\_const1);

\_\_m128i cr1 = \_mm\_mulhi\_epi16(crw, cr\_const1);

\_\_m128i rws = \_mm\_add\_epi16(cr0, yws);

\_\_m128i gwt = \_mm\_add\_epi16(cb0, yws);

\_\_m128i bws = \_mm\_add\_epi16(yws, cb1);

\_\_m128i gws = \_mm\_add\_epi16(gwt, cr1);

// descale

\_\_m128i rw = \_mm\_srai\_epi16(rws, 4);

\_\_m128i bw = \_mm\_srai\_epi16(bws, 4);

\_\_m128i gw = \_mm\_srai\_epi16(gws, 4);

// back to byte, set up for transpose

\_\_m128i brb = \_mm\_packus\_epi16(rw, bw);

\_\_m128i gxb = \_mm\_packus\_epi16(gw, xw);

// transpose to interleave channels

\_\_m128i t0 = \_mm\_unpacklo\_epi8(brb, gxb);

\_\_m128i t1 = \_mm\_unpackhi\_epi8(brb, gxb);

\_\_m128i o0 = \_mm\_unpacklo\_epi16(t0, t1);

\_\_m128i o1 = \_mm\_unpackhi\_epi16(t0, t1);

// store

\_mm\_storeu\_si128((\_\_m128i \*) (out + 0), o0);

\_mm\_storeu\_si128((\_\_m128i \*) (out + 16), o1);

out += 32;

}

}

#endif

#ifdef STBI\_NEON

// in this version, step=3 support would be easy to add. but is there demand?

if (step == 4) {

// this is a fairly straightforward implementation and not super-optimized.

uint8x8\_t signflip = vdup\_n\_u8(0x80);

int16x8\_t cr\_const0 = vdupq\_n\_s16( (short) ( 1.40200f\*4096.0f+0.5f));

int16x8\_t cr\_const1 = vdupq\_n\_s16( - (short) ( 0.71414f\*4096.0f+0.5f));

int16x8\_t cb\_const0 = vdupq\_n\_s16( - (short) ( 0.34414f\*4096.0f+0.5f));

int16x8\_t cb\_const1 = vdupq\_n\_s16( (short) ( 1.77200f\*4096.0f+0.5f));

for (; i+7 < count; i += 8) {

// load

uint8x8\_t y\_bytes = vld1\_u8(y + i);

uint8x8\_t cr\_bytes = vld1\_u8(pcr + i);

uint8x8\_t cb\_bytes = vld1\_u8(pcb + i);

int8x8\_t cr\_biased = vreinterpret\_s8\_u8(vsub\_u8(cr\_bytes, signflip));

int8x8\_t cb\_biased = vreinterpret\_s8\_u8(vsub\_u8(cb\_bytes, signflip));

// expand to s16

int16x8\_t yws = vreinterpretq\_s16\_u16(vshll\_n\_u8(y\_bytes, 4));

int16x8\_t crw = vshll\_n\_s8(cr\_biased, 7);

int16x8\_t cbw = vshll\_n\_s8(cb\_biased, 7);

// color transform

int16x8\_t cr0 = vqdmulhq\_s16(crw, cr\_const0);

int16x8\_t cb0 = vqdmulhq\_s16(cbw, cb\_const0);

int16x8\_t cr1 = vqdmulhq\_s16(crw, cr\_const1);

int16x8\_t cb1 = vqdmulhq\_s16(cbw, cb\_const1);

int16x8\_t rws = vaddq\_s16(yws, cr0);

int16x8\_t gws = vaddq\_s16(vaddq\_s16(yws, cb0), cr1);

int16x8\_t bws = vaddq\_s16(yws, cb1);

// undo scaling, round, convert to byte

uint8x8x4\_t o;

o.val[0] = vqrshrun\_n\_s16(rws, 4);

o.val[1] = vqrshrun\_n\_s16(gws, 4);

o.val[2] = vqrshrun\_n\_s16(bws, 4);

o.val[3] = vdup\_n\_u8(255);

// store, interleaving r/g/b/a

vst4\_u8(out, o);

out += 8\*4;

}

}

#endif

for (; i < count; ++i) {

int y\_fixed = (y[i] << 20) + (1<<19); // rounding

int r,g,b;

int cr = pcr[i] - 128;

int cb = pcb[i] - 128;

r = y\_fixed + cr\* stbi\_\_float2fixed(1.40200f);

g = y\_fixed + cr\*-stbi\_\_float2fixed(0.71414f) + ((cb\*-stbi\_\_float2fixed(0.34414f)) & 0xffff0000);

b = y\_fixed + cb\* stbi\_\_float2fixed(1.77200f);

r >>= 20;

g >>= 20;

b >>= 20;

if ((unsigned) r > 255) { if (r < 0) r = 0; else r = 255; }

if ((unsigned) g > 255) { if (g < 0) g = 0; else g = 255; }

if ((unsigned) b > 255) { if (b < 0) b = 0; else b = 255; }

out[0] = (stbi\_uc)r;

out[1] = (stbi\_uc)g;

out[2] = (stbi\_uc)b;

out[3] = 255;

out += step;

}

}

#endif

// set up the kernels

static void stbi\_\_setup\_jpeg(stbi\_\_jpeg \*j)

{

j->idct\_block\_kernel = stbi\_\_idct\_block;

j->YCbCr\_to\_RGB\_kernel = stbi\_\_YCbCr\_to\_RGB\_row;

j->resample\_row\_hv\_2\_kernel = stbi\_\_resample\_row\_hv\_2;

#ifdef STBI\_SSE2

if (stbi\_\_sse2\_available()) {

j->idct\_block\_kernel = stbi\_\_idct\_simd;

j->YCbCr\_to\_RGB\_kernel = stbi\_\_YCbCr\_to\_RGB\_simd;

j->resample\_row\_hv\_2\_kernel = stbi\_\_resample\_row\_hv\_2\_simd;

}

#endif

#ifdef STBI\_NEON

j->idct\_block\_kernel = stbi\_\_idct\_simd;

j->YCbCr\_to\_RGB\_kernel = stbi\_\_YCbCr\_to\_RGB\_simd;

j->resample\_row\_hv\_2\_kernel = stbi\_\_resample\_row\_hv\_2\_simd;

#endif

}

// clean up the temporary component buffers

static void stbi\_\_cleanup\_jpeg(stbi\_\_jpeg \*j)

{

stbi\_\_free\_jpeg\_components(j, j->s->img\_n, 0);

}

typedef struct

{

resample\_row\_func resample;

stbi\_uc \*line0,\*line1;

int hs,vs; // expansion factor in each axis

int w\_lores; // horizontal pixels pre-expansion

int ystep; // how far through vertical expansion we are

int ypos; // which pre-expansion row we're on

} stbi\_\_resample;

// fast 0..255 \* 0..255 => 0..255 rounded multiplication

static stbi\_uc stbi\_\_blinn\_8x8(stbi\_uc x, stbi\_uc y)

{

unsigned int t = x\*y + 128;

return (stbi\_uc) ((t + (t >>8)) >> 8);

}

static stbi\_uc \*load\_jpeg\_image(stbi\_\_jpeg \*z, int \*out\_x, int \*out\_y, int \*comp, int req\_comp)

{

int n, decode\_n, is\_rgb;

z->s->img\_n = 0; // make stbi\_\_cleanup\_jpeg safe

// validate req\_comp

if (req\_comp < 0 || req\_comp > 4) return stbi\_\_errpuc("bad req\_comp", "Internal error");

// load a jpeg image from whichever source, but leave in YCbCr format

if (!stbi\_\_decode\_jpeg\_image(z)) { stbi\_\_cleanup\_jpeg(z); return NULL; }

// determine actual number of components to generate

n = req\_comp ? req\_comp : z->s->img\_n >= 3 ? 3 : 1;

is\_rgb = z->s->img\_n == 3 && (z->rgb == 3 || (z->app14\_color\_transform == 0 && !z->jfif));

if (z->s->img\_n == 3 && n < 3 && !is\_rgb)

decode\_n = 1;

else

decode\_n = z->s->img\_n;

// resample and color-convert

{

int k;

unsigned int i,j;

stbi\_uc \*output;

stbi\_uc \*coutput[4];

stbi\_\_resample res\_comp[4];

for (k=0; k < decode\_n; ++k) {

stbi\_\_resample \*r = &res\_comp[k];

// allocate line buffer big enough for upsampling off the edges

// with upsample factor of 4

z->img\_comp[k].linebuf = (stbi\_uc \*) stbi\_\_malloc(z->s->img\_x + 3);

if (!z->img\_comp[k].linebuf) { stbi\_\_cleanup\_jpeg(z); return stbi\_\_errpuc("outofmem", "Out of memory"); }

r->hs = z->img\_h\_max / z->img\_comp[k].h;

r->vs = z->img\_v\_max / z->img\_comp[k].v;

r->ystep = r->vs >> 1;

r->w\_lores = (z->s->img\_x + r->hs-1) / r->hs;

r->ypos = 0;

r->line0 = r->line1 = z->img\_comp[k].data;

if (r->hs == 1 && r->vs == 1) r->resample = resample\_row\_1;

else if (r->hs == 1 && r->vs == 2) r->resample = stbi\_\_resample\_row\_v\_2;

else if (r->hs == 2 && r->vs == 1) r->resample = stbi\_\_resample\_row\_h\_2;

else if (r->hs == 2 && r->vs == 2) r->resample = z->resample\_row\_hv\_2\_kernel;

else r->resample = stbi\_\_resample\_row\_generic;

}

// can't error after this so, this is safe

output = (stbi\_uc \*) stbi\_\_malloc\_mad3(n, z->s->img\_x, z->s->img\_y, 1);

if (!output) { stbi\_\_cleanup\_jpeg(z); return stbi\_\_errpuc("outofmem", "Out of memory"); }

// now go ahead and resample

for (j=0; j < z->s->img\_y; ++j) {

stbi\_uc \*out = output + n \* z->s->img\_x \* j;

for (k=0; k < decode\_n; ++k) {

stbi\_\_resample \*r = &res\_comp[k];

int y\_bot = r->ystep >= (r->vs >> 1);

coutput[k] = r->resample(z->img\_comp[k].linebuf,

y\_bot ? r->line1 : r->line0,

y\_bot ? r->line0 : r->line1,

r->w\_lores, r->hs);

if (++r->ystep >= r->vs) {

r->ystep = 0;

r->line0 = r->line1;

if (++r->ypos < z->img\_comp[k].y)

r->line1 += z->img\_comp[k].w2;

}

}

if (n >= 3) {

stbi\_uc \*y = coutput[0];

if (z->s->img\_n == 3) {

if (is\_rgb) {

for (i=0; i < z->s->img\_x; ++i) {

out[0] = y[i];

out[1] = coutput[1][i];

out[2] = coutput[2][i];

out[3] = 255;

out += n;

}

} else {

z->YCbCr\_to\_RGB\_kernel(out, y, coutput[1], coutput[2], z->s->img\_x, n);

}

} else if (z->s->img\_n == 4) {

if (z->app14\_color\_transform == 0) { // CMYK

for (i=0; i < z->s->img\_x; ++i) {

stbi\_uc m = coutput[3][i];

out[0] = stbi\_\_blinn\_8x8(coutput[0][i], m);

out[1] = stbi\_\_blinn\_8x8(coutput[1][i], m);

out[2] = stbi\_\_blinn\_8x8(coutput[2][i], m);

out[3] = 255;

out += n;

}

} else if (z->app14\_color\_transform == 2) { // YCCK

z->YCbCr\_to\_RGB\_kernel(out, y, coutput[1], coutput[2], z->s->img\_x, n);

for (i=0; i < z->s->img\_x; ++i) {

stbi\_uc m = coutput[3][i];

out[0] = stbi\_\_blinn\_8x8(255 - out[0], m);

out[1] = stbi\_\_blinn\_8x8(255 - out[1], m);

out[2] = stbi\_\_blinn\_8x8(255 - out[2], m);

out += n;

}

} else { // YCbCr + alpha? Ignore the fourth channel for now

z->YCbCr\_to\_RGB\_kernel(out, y, coutput[1], coutput[2], z->s->img\_x, n);

}

} else

for (i=0; i < z->s->img\_x; ++i) {

out[0] = out[1] = out[2] = y[i];

out[3] = 255; // not used if n==3

out += n;

}

} else {

if (is\_rgb) {

if (n == 1)

for (i=0; i < z->s->img\_x; ++i)

\*out++ = stbi\_\_compute\_y(coutput[0][i], coutput[1][i], coutput[2][i]);

else {

for (i=0; i < z->s->img\_x; ++i, out += 2) {

out[0] = stbi\_\_compute\_y(coutput[0][i], coutput[1][i], coutput[2][i]);

out[1] = 255;

}

}

} else if (z->s->img\_n == 4 && z->app14\_color\_transform == 0) {

for (i=0; i < z->s->img\_x; ++i) {

stbi\_uc m = coutput[3][i];

stbi\_uc r = stbi\_\_blinn\_8x8(coutput[0][i], m);

stbi\_uc g = stbi\_\_blinn\_8x8(coutput[1][i], m);

stbi\_uc b = stbi\_\_blinn\_8x8(coutput[2][i], m);

out[0] = stbi\_\_compute\_y(r, g, b);

out[1] = 255;

out += n;

}

} else if (z->s->img\_n == 4 && z->app14\_color\_transform == 2) {

for (i=0; i < z->s->img\_x; ++i) {

out[0] = stbi\_\_blinn\_8x8(255 - coutput[0][i], coutput[3][i]);

out[1] = 255;

out += n;

}

} else {

stbi\_uc \*y = coutput[0];

if (n == 1)

for (i=0; i < z->s->img\_x; ++i) out[i] = y[i];

else

for (i=0; i < z->s->img\_x; ++i) \*out++ = y[i], \*out++ = 255;

}

}

}

stbi\_\_cleanup\_jpeg(z);

\*out\_x = z->s->img\_x;

\*out\_y = z->s->img\_y;

if (comp) \*comp = z->s->img\_n >= 3 ? 3 : 1; // report original components, not output

return output;

}

}

static void \*stbi\_\_jpeg\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri)

{

unsigned char\* result;

stbi\_\_jpeg\* j = (stbi\_\_jpeg\*) stbi\_\_malloc(sizeof(stbi\_\_jpeg));

STBI\_NOTUSED(ri);

j->s = s;

stbi\_\_setup\_jpeg(j);

result = load\_jpeg\_image(j, x,y,comp,req\_comp);

STBI\_FREE(j);

return result;

}

static int stbi\_\_jpeg\_test(stbi\_\_context \*s)

{

int r;

stbi\_\_jpeg\* j = (stbi\_\_jpeg\*)stbi\_\_malloc(sizeof(stbi\_\_jpeg));

j->s = s;

stbi\_\_setup\_jpeg(j);

r = stbi\_\_decode\_jpeg\_header(j, STBI\_\_SCAN\_type);

stbi\_\_rewind(s);

STBI\_FREE(j);

return r;

}

static int stbi\_\_jpeg\_info\_raw(stbi\_\_jpeg \*j, int \*x, int \*y, int \*comp)

{

if (!stbi\_\_decode\_jpeg\_header(j, STBI\_\_SCAN\_header)) {

stbi\_\_rewind( j->s );

return 0;

}

if (x) \*x = j->s->img\_x;

if (y) \*y = j->s->img\_y;

if (comp) \*comp = j->s->img\_n >= 3 ? 3 : 1;

return 1;

}

static int stbi\_\_jpeg\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

int result;

stbi\_\_jpeg\* j = (stbi\_\_jpeg\*) (stbi\_\_malloc(sizeof(stbi\_\_jpeg)));

j->s = s;

result = stbi\_\_jpeg\_info\_raw(j, x, y, comp);

STBI\_FREE(j);

return result;

}

#endif

// public domain zlib decode v0.2 Sean Barrett 2006-11-18

// simple implementation

// - all input must be provided in an upfront buffer

// - all output is written to a single output buffer (can malloc/realloc)

// performance

// - fast huffman

#ifndef STBI\_NO\_ZLIB

// fast-way is faster to check than jpeg huffman, but slow way is slower

#define STBI\_\_ZFAST\_BITS 9 // accelerate all cases in default tables

#define STBI\_\_ZFAST\_MASK ((1 << STBI\_\_ZFAST\_BITS) - 1)

// zlib-style huffman encoding

// (jpegs packs from left, zlib from right, so can't share code)

typedef struct

{

stbi\_\_uint16 fast[1 << STBI\_\_ZFAST\_BITS];

stbi\_\_uint16 firstcode[16];

int maxcode[17];

stbi\_\_uint16 firstsymbol[16];

stbi\_uc size[288];

stbi\_\_uint16 value[288];

} stbi\_\_zhuffman;

stbi\_inline static int stbi\_\_bitreverse16(int n)

{

n = ((n & 0xAAAA) >> 1) | ((n & 0x5555) << 1);

n = ((n & 0xCCCC) >> 2) | ((n & 0x3333) << 2);

n = ((n & 0xF0F0) >> 4) | ((n & 0x0F0F) << 4);

n = ((n & 0xFF00) >> 8) | ((n & 0x00FF) << 8);

return n;

}

stbi\_inline static int stbi\_\_bit\_reverse(int v, int bits)

{

STBI\_ASSERT(bits <= 16);

// to bit reverse n bits, reverse 16 and shift

// e.g. 11 bits, bit reverse and shift away 5

return stbi\_\_bitreverse16(v) >> (16-bits);

}

static int stbi\_\_zbuild\_huffman(stbi\_\_zhuffman \*z, const stbi\_uc \*sizelist, int num)

{

int i,k=0;

int code, next\_code[16], sizes[17];

// DEFLATE spec for generating codes

memset(sizes, 0, sizeof(sizes));

memset(z->fast, 0, sizeof(z->fast));

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

++sizes[sizelist[i]];

sizes[0] = 0;

for (i=1; i < 16; ++i)

if (sizes[i] > (1 << i))

return stbi\_\_err("bad sizes", "Corrupt PNG");

code = 0;

for (i=1; i < 16; ++i) {

next\_code[i] = code;

z->firstcode[i] = (stbi\_\_uint16) code;

z->firstsymbol[i] = (stbi\_\_uint16) k;

code = (code + sizes[i]);

if (sizes[i])

if (code-1 >= (1 << i)) return stbi\_\_err("bad codelengths","Corrupt PNG");

z->maxcode[i] = code << (16-i); // preshift for inner loop

code <<= 1;

k += sizes[i];

}

z->maxcode[16] = 0x10000; // sentinel

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

int s = sizelist[i];

if (s) {

int c = next\_code[s] - z->firstcode[s] + z->firstsymbol[s];

stbi\_\_uint16 fastv = (stbi\_\_uint16) ((s << 9) | i);

z->size [c] = (stbi\_uc ) s;

z->value[c] = (stbi\_\_uint16) i;

if (s <= STBI\_\_ZFAST\_BITS) {

int j = stbi\_\_bit\_reverse(next\_code[s],s);

while (j < (1 << STBI\_\_ZFAST\_BITS)) {

z->fast[j] = fastv;

j += (1 << s);

}

}

++next\_code[s];

}

}

return 1;

}

// zlib-from-memory implementation for PNG reading

// because PNG allows splitting the zlib stream arbitrarily,

// and it's annoying structurally to have PNG call ZLIB call PNG,

// we require PNG read all the IDATs and combine them into a single

// memory buffer

typedef struct

{

stbi\_uc \*zbuffer, \*zbuffer\_end;

int num\_bits;

stbi\_\_uint32 code\_buffer;

char \*zout;

char \*zout\_start;

char \*zout\_end;

int z\_expandable;

stbi\_\_zhuffman z\_length, z\_distance;

} stbi\_\_zbuf;

stbi\_inline static stbi\_uc stbi\_\_zget8(stbi\_\_zbuf \*z)

{

if (z->zbuffer >= z->zbuffer\_end) return 0;

return \*z->zbuffer++;

}

static void stbi\_\_fill\_bits(stbi\_\_zbuf \*z)

{

do {

STBI\_ASSERT(z->code\_buffer < (1U << z->num\_bits));

z->code\_buffer |= (unsigned int) stbi\_\_zget8(z) << z->num\_bits;

z->num\_bits += 8;

} while (z->num\_bits <= 24);

}

stbi\_inline static unsigned int stbi\_\_zreceive(stbi\_\_zbuf \*z, int n)

{

unsigned int k;

if (z->num\_bits < n) stbi\_\_fill\_bits(z);

k = z->code\_buffer & ((1 << n) - 1);

z->code\_buffer >>= n;

z->num\_bits -= n;

return k;

}

static int stbi\_\_zhuffman\_decode\_slowpath(stbi\_\_zbuf \*a, stbi\_\_zhuffman \*z)

{

int b,s,k;

// not resolved by fast table, so compute it the slow way

// use jpeg approach, which requires MSbits at top

k = stbi\_\_bit\_reverse(a->code\_buffer, 16);

for (s=STBI\_\_ZFAST\_BITS+1; ; ++s)

if (k < z->maxcode[s])

break;

if (s == 16) return -1; // invalid code!

// code size is s, so:

b = (k >> (16-s)) - z->firstcode[s] + z->firstsymbol[s];

STBI\_ASSERT(z->size[b] == s);

a->code\_buffer >>= s;

a->num\_bits -= s;

return z->value[b];

}

stbi\_inline static int stbi\_\_zhuffman\_decode(stbi\_\_zbuf \*a, stbi\_\_zhuffman \*z)

{

int b,s;

if (a->num\_bits < 16) stbi\_\_fill\_bits(a);

b = z->fast[a->code\_buffer & STBI\_\_ZFAST\_MASK];

if (b) {

s = b >> 9;

a->code\_buffer >>= s;

a->num\_bits -= s;

return b & 511;

}

return stbi\_\_zhuffman\_decode\_slowpath(a, z);

}

static int stbi\_\_zexpand(stbi\_\_zbuf \*z, char \*zout, int n) // need to make room for n bytes

{

char \*q;

int cur, limit, old\_limit;

z->zout = zout;

if (!z->z\_expandable) return stbi\_\_err("output buffer limit","Corrupt PNG");

cur = (int) (z->zout - z->zout\_start);

limit = old\_limit = (int) (z->zout\_end - z->zout\_start);

while (cur + n > limit)

limit \*= 2;

q = (char \*) STBI\_REALLOC\_SIZED(z->zout\_start, old\_limit, limit);

STBI\_NOTUSED(old\_limit);

if (q == NULL) return stbi\_\_err("outofmem", "Out of memory");

z->zout\_start = q;

z->zout = q + cur;

z->zout\_end = q + limit;

return 1;

}

static int stbi\_\_zlength\_base[31] = {

3,4,5,6,7,8,9,10,11,13,

15,17,19,23,27,31,35,43,51,59,

67,83,99,115,131,163,195,227,258,0,0 };

static int stbi\_\_zlength\_extra[31]=

{ 0,0,0,0,0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,0,0,0 };

static int stbi\_\_zdist\_base[32] = { 1,2,3,4,5,7,9,13,17,25,33,49,65,97,129,193,

257,385,513,769,1025,1537,2049,3073,4097,6145,8193,12289,16385,24577,0,0};

static int stbi\_\_zdist\_extra[32] =

{ 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};

static int stbi\_\_parse\_huffman\_block(stbi\_\_zbuf \*a)

{

char \*zout = a->zout;

for(;;) {

int z = stbi\_\_zhuffman\_decode(a, &a->z\_length);

if (z < 256) {

if (z < 0) return stbi\_\_err("bad huffman code","Corrupt PNG"); // error in huffman codes

if (zout >= a->zout\_end) {

if (!stbi\_\_zexpand(a, zout, 1)) return 0;

zout = a->zout;

}

\*zout++ = (char) z;

} else {

stbi\_uc \*p;

int len,dist;

if (z == 256) {

a->zout = zout;

return 1;

}

z -= 257;

len = stbi\_\_zlength\_base[z];

if (stbi\_\_zlength\_extra[z]) len += stbi\_\_zreceive(a, stbi\_\_zlength\_extra[z]);

z = stbi\_\_zhuffman\_decode(a, &a->z\_distance);

if (z < 0) return stbi\_\_err("bad huffman code","Corrupt PNG");

dist = stbi\_\_zdist\_base[z];

if (stbi\_\_zdist\_extra[z]) dist += stbi\_\_zreceive(a, stbi\_\_zdist\_extra[z]);

if (zout - a->zout\_start < dist) return stbi\_\_err("bad dist","Corrupt PNG");

if (zout + len > a->zout\_end) {

if (!stbi\_\_zexpand(a, zout, len)) return 0;

zout = a->zout;

}

p = (stbi\_uc \*) (zout - dist);

if (dist == 1) { // run of one byte; common in images.

stbi\_uc v = \*p;

if (len) { do \*zout++ = v; while (--len); }

} else {

if (len) { do \*zout++ = \*p++; while (--len); }

}

}

}

}

static int stbi\_\_compute\_huffman\_codes(stbi\_\_zbuf \*a)

{

static stbi\_uc length\_dezigzag[19] = { 16,17,18,0,8,7,9,6,10,5,11,4,12,3,13,2,14,1,15 };

stbi\_\_zhuffman z\_codelength;

stbi\_uc lencodes[286+32+137];//padding for maximum single op

stbi\_uc codelength\_sizes[19];

int i,n;

int hlit = stbi\_\_zreceive(a,5) + 257;

int hdist = stbi\_\_zreceive(a,5) + 1;

int hclen = stbi\_\_zreceive(a,4) + 4;

int ntot = hlit + hdist;

memset(codelength\_sizes, 0, sizeof(codelength\_sizes));

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

int s = stbi\_\_zreceive(a,3);

codelength\_sizes[length\_dezigzag[i]] = (stbi\_uc) s;

}

if (!stbi\_\_zbuild\_huffman(&z\_codelength, codelength\_sizes, 19)) return 0;

n = 0;

while (n < ntot) {

int c = stbi\_\_zhuffman\_decode(a, &z\_codelength);

if (c < 0 || c >= 19) return stbi\_\_err("bad codelengths", "Corrupt PNG");

if (c < 16)

lencodes[n++] = (stbi\_uc) c;

else {

stbi\_uc fill = 0;

if (c == 16) {

c = stbi\_\_zreceive(a,2)+3;

if (n == 0) return stbi\_\_err("bad codelengths", "Corrupt PNG");

fill = lencodes[n-1];

} else if (c == 17)

c = stbi\_\_zreceive(a,3)+3;

else {

STBI\_ASSERT(c == 18);

c = stbi\_\_zreceive(a,7)+11;

}

if (ntot - n < c) return stbi\_\_err("bad codelengths", "Corrupt PNG");

memset(lencodes+n, fill, c);

n += c;

}

}

if (n != ntot) return stbi\_\_err("bad codelengths","Corrupt PNG");

if (!stbi\_\_zbuild\_huffman(&a->z\_length, lencodes, hlit)) return 0;

if (!stbi\_\_zbuild\_huffman(&a->z\_distance, lencodes+hlit, hdist)) return 0;

return 1;

}

static int stbi\_\_parse\_uncompressed\_block(stbi\_\_zbuf \*a)

{

stbi\_uc header[4];

int len,nlen,k;

if (a->num\_bits & 7)

stbi\_\_zreceive(a, a->num\_bits & 7); // discard

// drain the bit-packed data into header

k = 0;

while (a->num\_bits > 0) {

header[k++] = (stbi\_uc) (a->code\_buffer & 255); // suppress MSVC run-time check

a->code\_buffer >>= 8;

a->num\_bits -= 8;

}

STBI\_ASSERT(a->num\_bits == 0);

// now fill header the normal way

while (k < 4)

header[k++] = stbi\_\_zget8(a);

len = header[1] \* 256 + header[0];

nlen = header[3] \* 256 + header[2];

if (nlen != (len ^ 0xffff)) return stbi\_\_err("zlib corrupt","Corrupt PNG");

if (a->zbuffer + len > a->zbuffer\_end) return stbi\_\_err("read past buffer","Corrupt PNG");

if (a->zout + len > a->zout\_end)

if (!stbi\_\_zexpand(a, a->zout, len)) return 0;

memcpy(a->zout, a->zbuffer, len);

a->zbuffer += len;

a->zout += len;

return 1;

}

static int stbi\_\_parse\_zlib\_header(stbi\_\_zbuf \*a)

{

int cmf = stbi\_\_zget8(a);

int cm = cmf & 15;

/\* int cinfo = cmf >> 4; \*/

int flg = stbi\_\_zget8(a);

if ((cmf\*256+flg) % 31 != 0) return stbi\_\_err("bad zlib header","Corrupt PNG"); // zlib spec

if (flg & 32) return stbi\_\_err("no preset dict","Corrupt PNG"); // preset dictionary not allowed in png

if (cm != 8) return stbi\_\_err("bad compression","Corrupt PNG"); // DEFLATE required for png

// window = 1 << (8 + cinfo)... but who cares, we fully buffer output

return 1;

}

static const stbi\_uc stbi\_\_zdefault\_length[288] =

{

8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,

8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,

8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,

8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,

8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,

9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9, 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,

9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9, 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,

9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9, 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,

7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7, 7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8

};

static const stbi\_uc stbi\_\_zdefault\_distance[32] =

{

5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5

};

/\*

Init algorithm:

{

int i; // use <= to match clearly with spec

for (i=0; i <= 143; ++i) stbi\_\_zdefault\_length[i] = 8;

for ( ; i <= 255; ++i) stbi\_\_zdefault\_length[i] = 9;

for ( ; i <= 279; ++i) stbi\_\_zdefault\_length[i] = 7;

for ( ; i <= 287; ++i) stbi\_\_zdefault\_length[i] = 8;

for (i=0; i <= 31; ++i) stbi\_\_zdefault\_distance[i] = 5;

}

\*/

static int stbi\_\_parse\_zlib(stbi\_\_zbuf \*a, int parse\_header)

{

int final, type;

if (parse\_header)

if (!stbi\_\_parse\_zlib\_header(a)) return 0;

a->num\_bits = 0;

a->code\_buffer = 0;

do {

final = stbi\_\_zreceive(a,1);

type = stbi\_\_zreceive(a,2);

if (type == 0) {

if (!stbi\_\_parse\_uncompressed\_block(a)) return 0;

} else if (type == 3) {

return 0;

} else {

if (type == 1) {

// use fixed code lengths

if (!stbi\_\_zbuild\_huffman(&a->z\_length , stbi\_\_zdefault\_length , 288)) return 0;

if (!stbi\_\_zbuild\_huffman(&a->z\_distance, stbi\_\_zdefault\_distance, 32)) return 0;

} else {

if (!stbi\_\_compute\_huffman\_codes(a)) return 0;

}

if (!stbi\_\_parse\_huffman\_block(a)) return 0;

}

} while (!final);

return 1;

}

static int stbi\_\_do\_zlib(stbi\_\_zbuf \*a, char \*obuf, int olen, int exp, int parse\_header)

{

a->zout\_start = obuf;

a->zout = obuf;

a->zout\_end = obuf + olen;

a->z\_expandable = exp;

return stbi\_\_parse\_zlib(a, parse\_header);

}

STBIDEF char \*stbi\_zlib\_decode\_malloc\_guesssize(const char \*buffer, int len, int initial\_size, int \*outlen)

{

stbi\_\_zbuf a;

char \*p = (char \*) stbi\_\_malloc(initial\_size);

if (p == NULL) return NULL;

a.zbuffer = (stbi\_uc \*) buffer;

a.zbuffer\_end = (stbi\_uc \*) buffer + len;

if (stbi\_\_do\_zlib(&a, p, initial\_size, 1, 1)) {

if (outlen) \*outlen = (int) (a.zout - a.zout\_start);

return a.zout\_start;

} else {

STBI\_FREE(a.zout\_start);

return NULL;

}

}

STBIDEF char \*stbi\_zlib\_decode\_malloc(char const \*buffer, int len, int \*outlen)

{

return stbi\_zlib\_decode\_malloc\_guesssize(buffer, len, 16384, outlen);

}

STBIDEF char \*stbi\_zlib\_decode\_malloc\_guesssize\_headerflag(const char \*buffer, int len, int initial\_size, int \*outlen, int parse\_header)

{

stbi\_\_zbuf a;

char \*p = (char \*) stbi\_\_malloc(initial\_size);

if (p == NULL) return NULL;

a.zbuffer = (stbi\_uc \*) buffer;

a.zbuffer\_end = (stbi\_uc \*) buffer + len;

if (stbi\_\_do\_zlib(&a, p, initial\_size, 1, parse\_header)) {

if (outlen) \*outlen = (int) (a.zout - a.zout\_start);

return a.zout\_start;

} else {

STBI\_FREE(a.zout\_start);

return NULL;

}

}

STBIDEF int stbi\_zlib\_decode\_buffer(char \*obuffer, int olen, char const \*ibuffer, int ilen)

{

stbi\_\_zbuf a;

a.zbuffer = (stbi\_uc \*) ibuffer;

a.zbuffer\_end = (stbi\_uc \*) ibuffer + ilen;

if (stbi\_\_do\_zlib(&a, obuffer, olen, 0, 1))

return (int) (a.zout - a.zout\_start);

else

return -1;

}

STBIDEF char \*stbi\_zlib\_decode\_noheader\_malloc(char const \*buffer, int len, int \*outlen)

{

stbi\_\_zbuf a;

char \*p = (char \*) stbi\_\_malloc(16384);

if (p == NULL) return NULL;

a.zbuffer = (stbi\_uc \*) buffer;

a.zbuffer\_end = (stbi\_uc \*) buffer+len;

if (stbi\_\_do\_zlib(&a, p, 16384, 1, 0)) {

if (outlen) \*outlen = (int) (a.zout - a.zout\_start);

return a.zout\_start;

} else {

STBI\_FREE(a.zout\_start);

return NULL;

}

}

STBIDEF int stbi\_zlib\_decode\_noheader\_buffer(char \*obuffer, int olen, const char \*ibuffer, int ilen)

{

stbi\_\_zbuf a;

a.zbuffer = (stbi\_uc \*) ibuffer;

a.zbuffer\_end = (stbi\_uc \*) ibuffer + ilen;

if (stbi\_\_do\_zlib(&a, obuffer, olen, 0, 0))

return (int) (a.zout - a.zout\_start);

else

return -1;

}

#endif

// public domain "baseline" PNG decoder v0.10 Sean Barrett 2006-11-18

// simple implementation

// - only 8-bit samples

// - no CRC checking

// - allocates lots of intermediate memory

// - avoids problem of streaming data between subsystems

// - avoids explicit window management

// performance

// - uses stb\_zlib, a PD zlib implementation with fast huffman decoding

#ifndef STBI\_NO\_PNG

typedef struct

{

stbi\_\_uint32 length;

stbi\_\_uint32 type;

} stbi\_\_pngchunk;

static stbi\_\_pngchunk stbi\_\_get\_chunk\_header(stbi\_\_context \*s)

{

stbi\_\_pngchunk c;

c.length = stbi\_\_get32be(s);

c.type = stbi\_\_get32be(s);

return c;

}

static int stbi\_\_check\_png\_header(stbi\_\_context \*s)

{

static stbi\_uc png\_sig[8] = { 137,80,78,71,13,10,26,10 };

int i;

for (i=0; i < 8; ++i)

if (stbi\_\_get8(s) != png\_sig[i]) return stbi\_\_err("bad png sig","Not a PNG");

return 1;

}

typedef struct

{

stbi\_\_context \*s;

stbi\_uc \*idata, \*expanded, \*out;

int depth;

} stbi\_\_png;

enum {

STBI\_\_F\_none=0,

STBI\_\_F\_sub=1,

STBI\_\_F\_up=2,

STBI\_\_F\_avg=3,

STBI\_\_F\_paeth=4,

// synthetic filters used for first scanline to avoid needing a dummy row of 0s

STBI\_\_F\_avg\_first,

STBI\_\_F\_paeth\_first

};

static stbi\_uc first\_row\_filter[5] =

{

STBI\_\_F\_none,

STBI\_\_F\_sub,

STBI\_\_F\_none,

STBI\_\_F\_avg\_first,

STBI\_\_F\_paeth\_first

};

static int stbi\_\_paeth(int a, int b, int c)

{

int p = a + b - c;

int pa = abs(p-a);

int pb = abs(p-b);

int pc = abs(p-c);

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

if (pb <= pc) return b;

return c;

}

static stbi\_uc stbi\_\_depth\_scale\_table[9] = { 0, 0xff, 0x55, 0, 0x11, 0,0,0, 0x01 };

// create the png data from post-deflated data

static int stbi\_\_create\_png\_image\_raw(stbi\_\_png \*a, stbi\_uc \*raw, stbi\_\_uint32 raw\_len, int out\_n, stbi\_\_uint32 x, stbi\_\_uint32 y, int depth, int color)

{

int bytes = (depth == 16? 2 : 1);

stbi\_\_context \*s = a->s;

stbi\_\_uint32 i,j,stride = x\*out\_n\*bytes;

stbi\_\_uint32 img\_len, img\_width\_bytes;

int k;

int img\_n = s->img\_n; // copy it into a local for later

int output\_bytes = out\_n\*bytes;

int filter\_bytes = img\_n\*bytes;

int width = x;

STBI\_ASSERT(out\_n == s->img\_n || out\_n == s->img\_n+1);

a->out = (stbi\_uc \*) stbi\_\_malloc\_mad3(x, y, output\_bytes, 0); // extra bytes to write off the end into

if (!a->out) return stbi\_\_err("outofmem", "Out of memory");

img\_width\_bytes = (((img\_n \* x \* depth) + 7) >> 3);

img\_len = (img\_width\_bytes + 1) \* y;

// we used to check for exact match between raw\_len and img\_len on non-interlaced PNGs,

// but issue #276 reported a PNG in the wild that had extra data at the end (all zeros),

// so just check for raw\_len < img\_len always.

if (raw\_len < img\_len) return stbi\_\_err("not enough pixels","Corrupt PNG");

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

stbi\_uc \*cur = a->out + stride\*j;

stbi\_uc \*prior;

int filter = \*raw++;

if (filter > 4)

return stbi\_\_err("invalid filter","Corrupt PNG");

if (depth < 8) {

STBI\_ASSERT(img\_width\_bytes <= x);

cur += x\*out\_n - img\_width\_bytes; // store output to the rightmost img\_len bytes, so we can decode in place

filter\_bytes = 1;

width = img\_width\_bytes;

}

prior = cur - stride; // bugfix: need to compute this after 'cur +=' computation above

// if first row, use special filter that doesn't sample previous row

if (j == 0) filter = first\_row\_filter[filter];

// handle first byte explicitly

for (k=0; k < filter\_bytes; ++k) {

switch (filter) {

case STBI\_\_F\_none : cur[k] = raw[k]; break;

case STBI\_\_F\_sub : cur[k] = raw[k]; break;

case STBI\_\_F\_up : cur[k] = STBI\_\_BYTECAST(raw[k] + prior[k]); break;

case STBI\_\_F\_avg : cur[k] = STBI\_\_BYTECAST(raw[k] + (prior[k]>>1)); break;

case STBI\_\_F\_paeth : cur[k] = STBI\_\_BYTECAST(raw[k] + stbi\_\_paeth(0,prior[k],0)); break;

case STBI\_\_F\_avg\_first : cur[k] = raw[k]; break;

case STBI\_\_F\_paeth\_first: cur[k] = raw[k]; break;

}

}

if (depth == 8) {

if (img\_n != out\_n)

cur[img\_n] = 255; // first pixel

raw += img\_n;

cur += out\_n;

prior += out\_n;

} else if (depth == 16) {

if (img\_n != out\_n) {

cur[filter\_bytes] = 255; // first pixel top byte

cur[filter\_bytes+1] = 255; // first pixel bottom byte

}

raw += filter\_bytes;

cur += output\_bytes;

prior += output\_bytes;

} else {

raw += 1;

cur += 1;

prior += 1;

}

// this is a little gross, so that we don't switch per-pixel or per-component

if (depth < 8 || img\_n == out\_n) {

int nk = (width - 1)\*filter\_bytes;

#define STBI\_\_CASE(f) \

case f: \

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

switch (filter) {

// "none" filter turns into a memcpy here; make that explicit.

case STBI\_\_F\_none: memcpy(cur, raw, nk); break;

STBI\_\_CASE(STBI\_\_F\_sub) { cur[k] = STBI\_\_BYTECAST(raw[k] + cur[k-filter\_bytes]); } break;

STBI\_\_CASE(STBI\_\_F\_up) { cur[k] = STBI\_\_BYTECAST(raw[k] + prior[k]); } break;

STBI\_\_CASE(STBI\_\_F\_avg) { cur[k] = STBI\_\_BYTECAST(raw[k] + ((prior[k] + cur[k-filter\_bytes])>>1)); } break;

STBI\_\_CASE(STBI\_\_F\_paeth) { cur[k] = STBI\_\_BYTECAST(raw[k] + stbi\_\_paeth(cur[k-filter\_bytes],prior[k],prior[k-filter\_bytes])); } break;

STBI\_\_CASE(STBI\_\_F\_avg\_first) { cur[k] = STBI\_\_BYTECAST(raw[k] + (cur[k-filter\_bytes] >> 1)); } break;

STBI\_\_CASE(STBI\_\_F\_paeth\_first) { cur[k] = STBI\_\_BYTECAST(raw[k] + stbi\_\_paeth(cur[k-filter\_bytes],0,0)); } break;

}

#undef STBI\_\_CASE

raw += nk;

} else {

STBI\_ASSERT(img\_n+1 == out\_n);

#define STBI\_\_CASE(f) \

case f: \

for (i=x-1; i >= 1; --i, cur[filter\_bytes]=255,raw+=filter\_bytes,cur+=output\_bytes,prior+=output\_bytes) \

for (k=0; k < filter\_bytes; ++k)

switch (filter) {

STBI\_\_CASE(STBI\_\_F\_none) { cur[k] = raw[k]; } break;

STBI\_\_CASE(STBI\_\_F\_sub) { cur[k] = STBI\_\_BYTECAST(raw[k] + cur[k- output\_bytes]); } break;

STBI\_\_CASE(STBI\_\_F\_up) { cur[k] = STBI\_\_BYTECAST(raw[k] + prior[k]); } break;

STBI\_\_CASE(STBI\_\_F\_avg) { cur[k] = STBI\_\_BYTECAST(raw[k] + ((prior[k] + cur[k- output\_bytes])>>1)); } break;

STBI\_\_CASE(STBI\_\_F\_paeth) { cur[k] = STBI\_\_BYTECAST(raw[k] + stbi\_\_paeth(cur[k- output\_bytes],prior[k],prior[k- output\_bytes])); } break;

STBI\_\_CASE(STBI\_\_F\_avg\_first) { cur[k] = STBI\_\_BYTECAST(raw[k] + (cur[k- output\_bytes] >> 1)); } break;

STBI\_\_CASE(STBI\_\_F\_paeth\_first) { cur[k] = STBI\_\_BYTECAST(raw[k] + stbi\_\_paeth(cur[k- output\_bytes],0,0)); } break;

}

#undef STBI\_\_CASE

// the loop above sets the high byte of the pixels' alpha, but for

// 16 bit png files we also need the low byte set. we'll do that here.

if (depth == 16) {

cur = a->out + stride\*j; // start at the beginning of the row again

for (i=0; i < x; ++i,cur+=output\_bytes) {

cur[filter\_bytes+1] = 255;

}

}

}

}

// we make a separate pass to expand bits to pixels; for performance,

// this could run two scanlines behind the above code, so it won't

// intefere with filtering but will still be in the cache.

if (depth < 8) {

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

stbi\_uc \*cur = a->out + stride\*j;

stbi\_uc \*in = a->out + stride\*j + x\*out\_n - img\_width\_bytes;

// unpack 1/2/4-bit into a 8-bit buffer. allows us to keep the common 8-bit path optimal at minimal cost for 1/2/4-bit

// png guarante byte alignment, if width is not multiple of 8/4/2 we'll decode dummy trailing data that will be skipped in the later loop

stbi\_uc scale = (color == 0) ? stbi\_\_depth\_scale\_table[depth] : 1; // scale grayscale values to 0..255 range

// note that the final byte might overshoot and write more data than desired.

// we can allocate enough data that this never writes out of memory, but it

// could also overwrite the next scanline. can it overwrite non-empty data

// on the next scanline? yes, consider 1-pixel-wide scanlines with 1-bit-per-pixel.

// so we need to explicitly clamp the final ones

if (depth == 4) {

for (k=x\*img\_n; k >= 2; k-=2, ++in) {

\*cur++ = scale \* ((\*in >> 4) );

\*cur++ = scale \* ((\*in ) & 0x0f);

}

if (k > 0) \*cur++ = scale \* ((\*in >> 4) );

} else if (depth == 2) {

for (k=x\*img\_n; k >= 4; k-=4, ++in) {

\*cur++ = scale \* ((\*in >> 6) );

\*cur++ = scale \* ((\*in >> 4) & 0x03);

\*cur++ = scale \* ((\*in >> 2) & 0x03);

\*cur++ = scale \* ((\*in ) & 0x03);

}

if (k > 0) \*cur++ = scale \* ((\*in >> 6) );

if (k > 1) \*cur++ = scale \* ((\*in >> 4) & 0x03);

if (k > 2) \*cur++ = scale \* ((\*in >> 2) & 0x03);

} else if (depth == 1) {

for (k=x\*img\_n; k >= 8; k-=8, ++in) {

\*cur++ = scale \* ((\*in >> 7) );

\*cur++ = scale \* ((\*in >> 6) & 0x01);

\*cur++ = scale \* ((\*in >> 5) & 0x01);

\*cur++ = scale \* ((\*in >> 4) & 0x01);

\*cur++ = scale \* ((\*in >> 3) & 0x01);

\*cur++ = scale \* ((\*in >> 2) & 0x01);

\*cur++ = scale \* ((\*in >> 1) & 0x01);

\*cur++ = scale \* ((\*in ) & 0x01);

}

if (k > 0) \*cur++ = scale \* ((\*in >> 7) );

if (k > 1) \*cur++ = scale \* ((\*in >> 6) & 0x01);

if (k > 2) \*cur++ = scale \* ((\*in >> 5) & 0x01);

if (k > 3) \*cur++ = scale \* ((\*in >> 4) & 0x01);

if (k > 4) \*cur++ = scale \* ((\*in >> 3) & 0x01);

if (k > 5) \*cur++ = scale \* ((\*in >> 2) & 0x01);

if (k > 6) \*cur++ = scale \* ((\*in >> 1) & 0x01);

}

if (img\_n != out\_n) {

int q;

// insert alpha = 255

cur = a->out + stride\*j;

if (img\_n == 1) {

for (q=x-1; q >= 0; --q) {

cur[q\*2+1] = 255;

cur[q\*2+0] = cur[q];

}

} else {

STBI\_ASSERT(img\_n == 3);

for (q=x-1; q >= 0; --q) {

cur[q\*4+3] = 255;

cur[q\*4+2] = cur[q\*3+2];

cur[q\*4+1] = cur[q\*3+1];

cur[q\*4+0] = cur[q\*3+0];

}

}

}

}

} else if (depth == 16) {

// force the image data from big-endian to platform-native.

// this is done in a separate pass due to the decoding relying

// on the data being untouched, but could probably be done

// per-line during decode if care is taken.

stbi\_uc \*cur = a->out;

stbi\_\_uint16 \*cur16 = (stbi\_\_uint16\*)cur;

for(i=0; i < x\*y\*out\_n; ++i,cur16++,cur+=2) {

\*cur16 = (cur[0] << 8) | cur[1];

}

}

return 1;

}

static int stbi\_\_create\_png\_image(stbi\_\_png \*a, stbi\_uc \*image\_data, stbi\_\_uint32 image\_data\_len, int out\_n, int depth, int color, int interlaced)

{

int bytes = (depth == 16 ? 2 : 1);

int out\_bytes = out\_n \* bytes;

stbi\_uc \*final;

int p;

if (!interlaced)

return stbi\_\_create\_png\_image\_raw(a, image\_data, image\_data\_len, out\_n, a->s->img\_x, a->s->img\_y, depth, color);

// de-interlacing

final = (stbi\_uc \*) stbi\_\_malloc\_mad3(a->s->img\_x, a->s->img\_y, out\_bytes, 0);

for (p=0; p < 7; ++p) {

int xorig[] = { 0,4,0,2,0,1,0 };

int yorig[] = { 0,0,4,0,2,0,1 };

int xspc[] = { 8,8,4,4,2,2,1 };

int yspc[] = { 8,8,8,4,4,2,2 };

int i,j,x,y;

// pass1\_x[4] = 0, pass1\_x[5] = 1, pass1\_x[12] = 1

x = (a->s->img\_x - xorig[p] + xspc[p]-1) / xspc[p];

y = (a->s->img\_y - yorig[p] + yspc[p]-1) / yspc[p];

if (x && y) {

stbi\_\_uint32 img\_len = ((((a->s->img\_n \* x \* depth) + 7) >> 3) + 1) \* y;

if (!stbi\_\_create\_png\_image\_raw(a, image\_data, image\_data\_len, out\_n, x, y, depth, color)) {

STBI\_FREE(final);

return 0;

}

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

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

int out\_y = j\*yspc[p]+yorig[p];

int out\_x = i\*xspc[p]+xorig[p];

memcpy(final + out\_y\*a->s->img\_x\*out\_bytes + out\_x\*out\_bytes,

a->out + (j\*x+i)\*out\_bytes, out\_bytes);

}

}

STBI\_FREE(a->out);

image\_data += img\_len;

image\_data\_len -= img\_len;

}

}

a->out = final;

return 1;

}

static int stbi\_\_compute\_transparency(stbi\_\_png \*z, stbi\_uc tc[3], int out\_n)

{

stbi\_\_context \*s = z->s;

stbi\_\_uint32 i, pixel\_count = s->img\_x \* s->img\_y;

stbi\_uc \*p = z->out;

// compute color-based transparency, assuming we've

// already got 255 as the alpha value in the output

STBI\_ASSERT(out\_n == 2 || out\_n == 4);

if (out\_n == 2) {

for (i=0; i < pixel\_count; ++i) {

p[1] = (p[0] == tc[0] ? 0 : 255);

p += 2;

}

} else {

for (i=0; i < pixel\_count; ++i) {

if (p[0] == tc[0] && p[1] == tc[1] && p[2] == tc[2])

p[3] = 0;

p += 4;

}

}

return 1;

}

static int stbi\_\_compute\_transparency16(stbi\_\_png \*z, stbi\_\_uint16 tc[3], int out\_n)

{

stbi\_\_context \*s = z->s;

stbi\_\_uint32 i, pixel\_count = s->img\_x \* s->img\_y;

stbi\_\_uint16 \*p = (stbi\_\_uint16\*) z->out;

// compute color-based transparency, assuming we've

// already got 65535 as the alpha value in the output

STBI\_ASSERT(out\_n == 2 || out\_n == 4);

if (out\_n == 2) {

for (i = 0; i < pixel\_count; ++i) {

p[1] = (p[0] == tc[0] ? 0 : 65535);

p += 2;

}

} else {

for (i = 0; i < pixel\_count; ++i) {

if (p[0] == tc[0] && p[1] == tc[1] && p[2] == tc[2])

p[3] = 0;

p += 4;

}

}

return 1;

}

static int stbi\_\_expand\_png\_palette(stbi\_\_png \*a, stbi\_uc \*palette, int len, int pal\_img\_n)

{

stbi\_\_uint32 i, pixel\_count = a->s->img\_x \* a->s->img\_y;

stbi\_uc \*p, \*temp\_out, \*orig = a->out;

p = (stbi\_uc \*) stbi\_\_malloc\_mad2(pixel\_count, pal\_img\_n, 0);

if (p == NULL) return stbi\_\_err("outofmem", "Out of memory");

// between here and free(out) below, exitting would leak

temp\_out = p;

if (pal\_img\_n == 3) {

for (i=0; i < pixel\_count; ++i) {

int n = orig[i]\*4;

p[0] = palette[n ];

p[1] = palette[n+1];

p[2] = palette[n+2];

p += 3;

}

} else {

for (i=0; i < pixel\_count; ++i) {

int n = orig[i]\*4;

p[0] = palette[n ];

p[1] = palette[n+1];

p[2] = palette[n+2];

p[3] = palette[n+3];

p += 4;

}

}

STBI\_FREE(a->out);

a->out = temp\_out;

STBI\_NOTUSED(len);

return 1;

}

static int stbi\_\_unpremultiply\_on\_load = 0;

static int stbi\_\_de\_iphone\_flag = 0;

STBIDEF void stbi\_set\_unpremultiply\_on\_load(int flag\_true\_if\_should\_unpremultiply)

{

stbi\_\_unpremultiply\_on\_load = flag\_true\_if\_should\_unpremultiply;

}

STBIDEF void stbi\_convert\_iphone\_png\_to\_rgb(int flag\_true\_if\_should\_convert)

{

stbi\_\_de\_iphone\_flag = flag\_true\_if\_should\_convert;

}

static void stbi\_\_de\_iphone(stbi\_\_png \*z)

{

stbi\_\_context \*s = z->s;

stbi\_\_uint32 i, pixel\_count = s->img\_x \* s->img\_y;

stbi\_uc \*p = z->out;

if (s->img\_out\_n == 3) { // convert bgr to rgb

for (i=0; i < pixel\_count; ++i) {

stbi\_uc t = p[0];

p[0] = p[2];

p[2] = t;

p += 3;

}

} else {

STBI\_ASSERT(s->img\_out\_n == 4);

if (stbi\_\_unpremultiply\_on\_load) {

// convert bgr to rgb and unpremultiply

for (i=0; i < pixel\_count; ++i) {

stbi\_uc a = p[3];

stbi\_uc t = p[0];

if (a) {

stbi\_uc half = a / 2;

p[0] = (p[2] \* 255 + half) / a;

p[1] = (p[1] \* 255 + half) / a;

p[2] = ( t \* 255 + half) / a;

} else {

p[0] = p[2];

p[2] = t;

}

p += 4;

}

} else {

// convert bgr to rgb

for (i=0; i < pixel\_count; ++i) {

stbi\_uc t = p[0];

p[0] = p[2];

p[2] = t;

p += 4;

}

}

}

}

#define STBI\_\_PNG\_TYPE(a,b,c,d) (((a) << 24) + ((b) << 16) + ((c) << 8) + (d))

static int stbi\_\_parse\_png\_file(stbi\_\_png \*z, int scan, int req\_comp)

{

stbi\_uc palette[1024], pal\_img\_n=0;

stbi\_uc has\_trans=0, tc[3];

stbi\_\_uint16 tc16[3];

stbi\_\_uint32 ioff=0, idata\_limit=0, i, pal\_len=0;

int first=1,k,interlace=0, color=0, is\_iphone=0;

stbi\_\_context \*s = z->s;

z->expanded = NULL;

z->idata = NULL;

z->out = NULL;

if (!stbi\_\_check\_png\_header(s)) return 0;

if (scan == STBI\_\_SCAN\_type) return 1;

for (;;) {

stbi\_\_pngchunk c = stbi\_\_get\_chunk\_header(s);

switch (c.type) {

case STBI\_\_PNG\_TYPE('C','g','B','I'):

is\_iphone = 1;

stbi\_\_skip(s, c.length);

break;

case STBI\_\_PNG\_TYPE('I','H','D','R'): {

int comp,filter;

if (!first) return stbi\_\_err("multiple IHDR","Corrupt PNG");

first = 0;

if (c.length != 13) return stbi\_\_err("bad IHDR len","Corrupt PNG");

s->img\_x = stbi\_\_get32be(s); if (s->img\_x > (1 << 24)) return stbi\_\_err("too large","Very large image (corrupt?)");

s->img\_y = stbi\_\_get32be(s); if (s->img\_y > (1 << 24)) return stbi\_\_err("too large","Very large image (corrupt?)");

z->depth = stbi\_\_get8(s); if (z->depth != 1 && z->depth != 2 && z->depth != 4 && z->depth != 8 && z->depth != 16) return stbi\_\_err("1/2/4/8/16-bit only","PNG not supported: 1/2/4/8/16-bit only");

color = stbi\_\_get8(s); if (color > 6) return stbi\_\_err("bad ctype","Corrupt PNG");

if (color == 3 && z->depth == 16) return stbi\_\_err("bad ctype","Corrupt PNG");

if (color == 3) pal\_img\_n = 3; else if (color & 1) return stbi\_\_err("bad ctype","Corrupt PNG");

comp = stbi\_\_get8(s); if (comp) return stbi\_\_err("bad comp method","Corrupt PNG");

filter= stbi\_\_get8(s); if (filter) return stbi\_\_err("bad filter method","Corrupt PNG");

interlace = stbi\_\_get8(s); if (interlace>1) return stbi\_\_err("bad interlace method","Corrupt PNG");

if (!s->img\_x || !s->img\_y) return stbi\_\_err("0-pixel image","Corrupt PNG");

if (!pal\_img\_n) {

s->img\_n = (color & 2 ? 3 : 1) + (color & 4 ? 1 : 0);

if ((1 << 30) / s->img\_x / s->img\_n < s->img\_y) return stbi\_\_err("too large", "Image too large to decode");

if (scan == STBI\_\_SCAN\_header) return 1;

} else {

// if paletted, then pal\_n is our final components, and

// img\_n is # components to decompress/filter.

s->img\_n = 1;

if ((1 << 30) / s->img\_x / 4 < s->img\_y) return stbi\_\_err("too large","Corrupt PNG");

// if SCAN\_header, have to scan to see if we have a tRNS

}

break;

}

case STBI\_\_PNG\_TYPE('P','L','T','E'): {

if (first) return stbi\_\_err("first not IHDR", "Corrupt PNG");

if (c.length > 256\*3) return stbi\_\_err("invalid PLTE","Corrupt PNG");

pal\_len = c.length / 3;

if (pal\_len \* 3 != c.length) return stbi\_\_err("invalid PLTE","Corrupt PNG");

for (i=0; i < pal\_len; ++i) {

palette[i\*4+0] = stbi\_\_get8(s);

palette[i\*4+1] = stbi\_\_get8(s);

palette[i\*4+2] = stbi\_\_get8(s);

palette[i\*4+3] = 255;

}

break;

}

case STBI\_\_PNG\_TYPE('t','R','N','S'): {

if (first) return stbi\_\_err("first not IHDR", "Corrupt PNG");

if (z->idata) return stbi\_\_err("tRNS after IDAT","Corrupt PNG");

if (pal\_img\_n) {

if (scan == STBI\_\_SCAN\_header) { s->img\_n = 4; return 1; }

if (pal\_len == 0) return stbi\_\_err("tRNS before PLTE","Corrupt PNG");

if (c.length > pal\_len) return stbi\_\_err("bad tRNS len","Corrupt PNG");

pal\_img\_n = 4;

for (i=0; i < c.length; ++i)

palette[i\*4+3] = stbi\_\_get8(s);

} else {

if (!(s->img\_n & 1)) return stbi\_\_err("tRNS with alpha","Corrupt PNG");

if (c.length != (stbi\_\_uint32) s->img\_n\*2) return stbi\_\_err("bad tRNS len","Corrupt PNG");

has\_trans = 1;

if (z->depth == 16) {

for (k = 0; k < s->img\_n; ++k) tc16[k] = (stbi\_\_uint16)stbi\_\_get16be(s); // copy the values as-is

} else {

for (k = 0; k < s->img\_n; ++k) tc[k] = (stbi\_uc)(stbi\_\_get16be(s) & 255) \* stbi\_\_depth\_scale\_table[z->depth]; // non 8-bit images will be larger

}

}

break;

}

case STBI\_\_PNG\_TYPE('I','D','A','T'): {

if (first) return stbi\_\_err("first not IHDR", "Corrupt PNG");

if (pal\_img\_n && !pal\_len) return stbi\_\_err("no PLTE","Corrupt PNG");

if (scan == STBI\_\_SCAN\_header) { s->img\_n = pal\_img\_n; return 1; }

if ((int)(ioff + c.length) < (int)ioff) return 0;

if (ioff + c.length > idata\_limit) {

stbi\_\_uint32 idata\_limit\_old = idata\_limit;

stbi\_uc \*p;

if (idata\_limit == 0) idata\_limit = c.length > 4096 ? c.length : 4096;

while (ioff + c.length > idata\_limit)

idata\_limit \*= 2;

STBI\_NOTUSED(idata\_limit\_old);

p = (stbi\_uc \*) STBI\_REALLOC\_SIZED(z->idata, idata\_limit\_old, idata\_limit); if (p == NULL) return stbi\_\_err("outofmem", "Out of memory");

z->idata = p;

}

if (!stbi\_\_getn(s, z->idata+ioff,c.length)) return stbi\_\_err("outofdata","Corrupt PNG");

ioff += c.length;

break;

}

case STBI\_\_PNG\_TYPE('I','E','N','D'): {

stbi\_\_uint32 raw\_len, bpl;

if (first) return stbi\_\_err("first not IHDR", "Corrupt PNG");

if (scan != STBI\_\_SCAN\_load) return 1;

if (z->idata == NULL) return stbi\_\_err("no IDAT","Corrupt PNG");

// initial guess for decoded data size to avoid unnecessary reallocs

bpl = (s->img\_x \* z->depth + 7) / 8; // bytes per line, per component

raw\_len = bpl \* s->img\_y \* s->img\_n /\* pixels \*/ + s->img\_y /\* filter mode per row \*/;

z->expanded = (stbi\_uc \*) stbi\_zlib\_decode\_malloc\_guesssize\_headerflag((char \*) z->idata, ioff, raw\_len, (int \*) &raw\_len, !is\_iphone);

if (z->expanded == NULL) return 0; // zlib should set error

STBI\_FREE(z->idata); z->idata = NULL;

if ((req\_comp == s->img\_n+1 && req\_comp != 3 && !pal\_img\_n) || has\_trans)

s->img\_out\_n = s->img\_n+1;

else

s->img\_out\_n = s->img\_n;

if (!stbi\_\_create\_png\_image(z, z->expanded, raw\_len, s->img\_out\_n, z->depth, color, interlace)) return 0;

if (has\_trans) {

if (z->depth == 16) {

if (!stbi\_\_compute\_transparency16(z, tc16, s->img\_out\_n)) return 0;

} else {

if (!stbi\_\_compute\_transparency(z, tc, s->img\_out\_n)) return 0;

}

}

if (is\_iphone && stbi\_\_de\_iphone\_flag && s->img\_out\_n > 2)

stbi\_\_de\_iphone(z);

if (pal\_img\_n) {

// pal\_img\_n == 3 or 4

s->img\_n = pal\_img\_n; // record the actual colors we had

s->img\_out\_n = pal\_img\_n;

if (req\_comp >= 3) s->img\_out\_n = req\_comp;

if (!stbi\_\_expand\_png\_palette(z, palette, pal\_len, s->img\_out\_n))

return 0;

} else if (has\_trans) {

// non-paletted image with tRNS -> source image has (constant) alpha

++s->img\_n;

}

STBI\_FREE(z->expanded); z->expanded = NULL;

return 1;

}

default:

// if critical, fail

if (first) return stbi\_\_err("first not IHDR", "Corrupt PNG");

if ((c.type & (1 << 29)) == 0) {

#ifndef STBI\_NO\_FAILURE\_STRINGS

// not threadsafe

static char invalid\_chunk[] = "XXXX PNG chunk not known";

invalid\_chunk[0] = STBI\_\_BYTECAST(c.type >> 24);

invalid\_chunk[1] = STBI\_\_BYTECAST(c.type >> 16);

invalid\_chunk[2] = STBI\_\_BYTECAST(c.type >> 8);

invalid\_chunk[3] = STBI\_\_BYTECAST(c.type >> 0);

#endif

return stbi\_\_err(invalid\_chunk, "PNG not supported: unknown PNG chunk type");

}

stbi\_\_skip(s, c.length);

break;

}

// end of PNG chunk, read and skip CRC

stbi\_\_get32be(s);

}

}

static void \*stbi\_\_do\_png(stbi\_\_png \*p, int \*x, int \*y, int \*n, int req\_comp, stbi\_\_result\_info \*ri)

{

void \*result=NULL;

if (req\_comp < 0 || req\_comp > 4) return stbi\_\_errpuc("bad req\_comp", "Internal error");

if (stbi\_\_parse\_png\_file(p, STBI\_\_SCAN\_load, req\_comp)) {

if (p->depth < 8)

ri->bits\_per\_channel = 8;

else

ri->bits\_per\_channel = p->depth;

result = p->out;

p->out = NULL;

if (req\_comp && req\_comp != p->s->img\_out\_n) {

if (ri->bits\_per\_channel == 8)

result = stbi\_\_convert\_format((unsigned char \*) result, p->s->img\_out\_n, req\_comp, p->s->img\_x, p->s->img\_y);

else

result = stbi\_\_convert\_format16((stbi\_\_uint16 \*) result, p->s->img\_out\_n, req\_comp, p->s->img\_x, p->s->img\_y);

p->s->img\_out\_n = req\_comp;

if (result == NULL) return result;

}

\*x = p->s->img\_x;

\*y = p->s->img\_y;

if (n) \*n = p->s->img\_n;

}

STBI\_FREE(p->out); p->out = NULL;

STBI\_FREE(p->expanded); p->expanded = NULL;

STBI\_FREE(p->idata); p->idata = NULL;

return result;

}

static void \*stbi\_\_png\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri)

{

stbi\_\_png p;

p.s = s;

return stbi\_\_do\_png(&p, x,y,comp,req\_comp, ri);

}

static int stbi\_\_png\_test(stbi\_\_context \*s)

{

int r;

r = stbi\_\_check\_png\_header(s);

stbi\_\_rewind(s);

return r;

}

static int stbi\_\_png\_info\_raw(stbi\_\_png \*p, int \*x, int \*y, int \*comp)

{

if (!stbi\_\_parse\_png\_file(p, STBI\_\_SCAN\_header, 0)) {

stbi\_\_rewind( p->s );

return 0;

}

if (x) \*x = p->s->img\_x;

if (y) \*y = p->s->img\_y;

if (comp) \*comp = p->s->img\_n;

return 1;

}

static int stbi\_\_png\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

stbi\_\_png p;

p.s = s;

return stbi\_\_png\_info\_raw(&p, x, y, comp);

}

#endif

// Microsoft/Windows BMP image

#ifndef STBI\_NO\_BMP

static int stbi\_\_bmp\_test\_raw(stbi\_\_context \*s)

{

int r;

int sz;

if (stbi\_\_get8(s) != 'B') return 0;

if (stbi\_\_get8(s) != 'M') return 0;

stbi\_\_get32le(s); // discard filesize

stbi\_\_get16le(s); // discard reserved

stbi\_\_get16le(s); // discard reserved

stbi\_\_get32le(s); // discard data offset

sz = stbi\_\_get32le(s);

r = (sz == 12 || sz == 40 || sz == 56 || sz == 108 || sz == 124);

return r;

}

static int stbi\_\_bmp\_test(stbi\_\_context \*s)

{

int r = stbi\_\_bmp\_test\_raw(s);

stbi\_\_rewind(s);

return r;

}

// returns 0..31 for the highest set bit

static int stbi\_\_high\_bit(unsigned int z)

{

int n=0;

if (z == 0) return -1;

if (z >= 0x10000) n += 16, z >>= 16;

if (z >= 0x00100) n += 8, z >>= 8;

if (z >= 0x00010) n += 4, z >>= 4;

if (z >= 0x00004) n += 2, z >>= 2;

if (z >= 0x00002) n += 1, z >>= 1;

return n;

}

static int stbi\_\_bitcount(unsigned int a)

{

a = (a & 0x55555555) + ((a >> 1) & 0x55555555); // max 2

a = (a & 0x33333333) + ((a >> 2) & 0x33333333); // max 4

a = (a + (a >> 4)) & 0x0f0f0f0f; // max 8 per 4, now 8 bits

a = (a + (a >> 8)); // max 16 per 8 bits

a = (a + (a >> 16)); // max 32 per 8 bits

return a & 0xff;

}

static int stbi\_\_shiftsigned(int v, int shift, int bits)

{

int result;

int z=0;

if (shift < 0) v <<= -shift;

else v >>= shift;

result = v;

z = bits;

while (z < 8) {

result += v >> z;

z += bits;

}

return result;

}

typedef struct

{

int bpp, offset, hsz;

unsigned int mr,mg,mb,ma, all\_a;

} stbi\_\_bmp\_data;

static void \*stbi\_\_bmp\_parse\_header(stbi\_\_context \*s, stbi\_\_bmp\_data \*info)

{

int hsz;

if (stbi\_\_get8(s) != 'B' || stbi\_\_get8(s) != 'M') return stbi\_\_errpuc("not BMP", "Corrupt BMP");

stbi\_\_get32le(s); // discard filesize

stbi\_\_get16le(s); // discard reserved

stbi\_\_get16le(s); // discard reserved

info->offset = stbi\_\_get32le(s);

info->hsz = hsz = stbi\_\_get32le(s);

info->mr = info->mg = info->mb = info->ma = 0;

if (hsz != 12 && hsz != 40 && hsz != 56 && hsz != 108 && hsz != 124) return stbi\_\_errpuc("unknown BMP", "BMP type not supported: unknown");

if (hsz == 12) {

s->img\_x = stbi\_\_get16le(s);

s->img\_y = stbi\_\_get16le(s);

} else {

s->img\_x = stbi\_\_get32le(s);

s->img\_y = stbi\_\_get32le(s);

}

if (stbi\_\_get16le(s) != 1) return stbi\_\_errpuc("bad BMP", "bad BMP");

info->bpp = stbi\_\_get16le(s);

if (info->bpp == 1) return stbi\_\_errpuc("monochrome", "BMP type not supported: 1-bit");

if (hsz != 12) {

int compress = stbi\_\_get32le(s);

if (compress == 1 || compress == 2) return stbi\_\_errpuc("BMP RLE", "BMP type not supported: RLE");

stbi\_\_get32le(s); // discard sizeof

stbi\_\_get32le(s); // discard hres

stbi\_\_get32le(s); // discard vres

stbi\_\_get32le(s); // discard colorsused

stbi\_\_get32le(s); // discard max important

if (hsz == 40 || hsz == 56) {

if (hsz == 56) {

stbi\_\_get32le(s);

stbi\_\_get32le(s);

stbi\_\_get32le(s);

stbi\_\_get32le(s);

}

if (info->bpp == 16 || info->bpp == 32) {

if (compress == 0) {

if (info->bpp == 32) {

info->mr = 0xffu << 16;

info->mg = 0xffu << 8;

info->mb = 0xffu << 0;

info->ma = 0xffu << 24;

info->all\_a = 0; // if all\_a is 0 at end, then we loaded alpha channel but it was all 0

} else {

info->mr = 31u << 10;

info->mg = 31u << 5;

info->mb = 31u << 0;

}

} else if (compress == 3) {

info->mr = stbi\_\_get32le(s);

info->mg = stbi\_\_get32le(s);

info->mb = stbi\_\_get32le(s);

// not documented, but generated by photoshop and handled by mspaint

if (info->mr == info->mg && info->mg == info->mb) {

// ?!?!?

return stbi\_\_errpuc("bad BMP", "bad BMP");

}

} else

return stbi\_\_errpuc("bad BMP", "bad BMP");

}

} else {

int i;

if (hsz != 108 && hsz != 124)

return stbi\_\_errpuc("bad BMP", "bad BMP");

info->mr = stbi\_\_get32le(s);

info->mg = stbi\_\_get32le(s);

info->mb = stbi\_\_get32le(s);

info->ma = stbi\_\_get32le(s);

stbi\_\_get32le(s); // discard color space

for (i=0; i < 12; ++i)

stbi\_\_get32le(s); // discard color space parameters

if (hsz == 124) {

stbi\_\_get32le(s); // discard rendering intent

stbi\_\_get32le(s); // discard offset of profile data

stbi\_\_get32le(s); // discard size of profile data

stbi\_\_get32le(s); // discard reserved

}

}

}

return (void \*) 1;

}

static void \*stbi\_\_bmp\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri)

{

stbi\_uc \*out;

unsigned int mr=0,mg=0,mb=0,ma=0, all\_a;

stbi\_uc pal[256][4];

int psize=0,i,j,width;

int flip\_vertically, pad, target;

stbi\_\_bmp\_data info;

STBI\_NOTUSED(ri);

info.all\_a = 255;

if (stbi\_\_bmp\_parse\_header(s, &info) == NULL)

return NULL; // error code already set

flip\_vertically = ((int) s->img\_y) > 0;

s->img\_y = abs((int) s->img\_y);

mr = info.mr;

mg = info.mg;

mb = info.mb;

ma = info.ma;

all\_a = info.all\_a;

if (info.hsz == 12) {

if (info.bpp < 24)

psize = (info.offset - 14 - 24) / 3;

} else {

if (info.bpp < 16)

psize = (info.offset - 14 - info.hsz) >> 2;

}

s->img\_n = ma ? 4 : 3;

if (req\_comp && req\_comp >= 3) // we can directly decode 3 or 4

target = req\_comp;

else

target = s->img\_n; // if they want monochrome, we'll post-convert

// sanity-check size

if (!stbi\_\_mad3sizes\_valid(target, s->img\_x, s->img\_y, 0))

return stbi\_\_errpuc("too large", "Corrupt BMP");

out = (stbi\_uc \*) stbi\_\_malloc\_mad3(target, s->img\_x, s->img\_y, 0);

if (!out) return stbi\_\_errpuc("outofmem", "Out of memory");

if (info.bpp < 16) {

int z=0;

if (psize == 0 || psize > 256) { STBI\_FREE(out); return stbi\_\_errpuc("invalid", "Corrupt BMP"); }

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

pal[i][2] = stbi\_\_get8(s);

pal[i][1] = stbi\_\_get8(s);

pal[i][0] = stbi\_\_get8(s);

if (info.hsz != 12) stbi\_\_get8(s);

pal[i][3] = 255;

}

stbi\_\_skip(s, info.offset - 14 - info.hsz - psize \* (info.hsz == 12 ? 3 : 4));

if (info.bpp == 4) width = (s->img\_x + 1) >> 1;

else if (info.bpp == 8) width = s->img\_x;

else { STBI\_FREE(out); return stbi\_\_errpuc("bad bpp", "Corrupt BMP"); }

pad = (-width)&3;

for (j=0; j < (int) s->img\_y; ++j) {

for (i=0; i < (int) s->img\_x; i += 2) {

int v=stbi\_\_get8(s),v2=0;

if (info.bpp == 4) {

v2 = v & 15;

v >>= 4;

}

out[z++] = pal[v][0];

out[z++] = pal[v][1];

out[z++] = pal[v][2];

if (target == 4) out[z++] = 255;

if (i+1 == (int) s->img\_x) break;

v = (info.bpp == 8) ? stbi\_\_get8(s) : v2;

out[z++] = pal[v][0];

out[z++] = pal[v][1];

out[z++] = pal[v][2];

if (target == 4) out[z++] = 255;

}

stbi\_\_skip(s, pad);

}

} else {

int rshift=0,gshift=0,bshift=0,ashift=0,rcount=0,gcount=0,bcount=0,acount=0;

int z = 0;

int easy=0;

stbi\_\_skip(s, info.offset - 14 - info.hsz);

if (info.bpp == 24) width = 3 \* s->img\_x;

else if (info.bpp == 16) width = 2\*s->img\_x;

else /\* bpp = 32 and pad = 0 \*/ width=0;

pad = (-width) & 3;

if (info.bpp == 24) {

easy = 1;

} else if (info.bpp == 32) {

if (mb == 0xff && mg == 0xff00 && mr == 0x00ff0000 && ma == 0xff000000)

easy = 2;

}

if (!easy) {

if (!mr || !mg || !mb) { STBI\_FREE(out); return stbi\_\_errpuc("bad masks", "Corrupt BMP"); }

// right shift amt to put high bit in position #7

rshift = stbi\_\_high\_bit(mr)-7; rcount = stbi\_\_bitcount(mr);

gshift = stbi\_\_high\_bit(mg)-7; gcount = stbi\_\_bitcount(mg);

bshift = stbi\_\_high\_bit(mb)-7; bcount = stbi\_\_bitcount(mb);

ashift = stbi\_\_high\_bit(ma)-7; acount = stbi\_\_bitcount(ma);

}

for (j=0; j < (int) s->img\_y; ++j) {

if (easy) {

for (i=0; i < (int) s->img\_x; ++i) {

unsigned char a;

out[z+2] = stbi\_\_get8(s);

out[z+1] = stbi\_\_get8(s);

out[z+0] = stbi\_\_get8(s);

z += 3;

a = (easy == 2 ? stbi\_\_get8(s) : 255);

all\_a |= a;

if (target == 4) out[z++] = a;

}

} else {

int bpp = info.bpp;

for (i=0; i < (int) s->img\_x; ++i) {

stbi\_\_uint32 v = (bpp == 16 ? (stbi\_\_uint32) stbi\_\_get16le(s) : stbi\_\_get32le(s));

int a;

out[z++] = STBI\_\_BYTECAST(stbi\_\_shiftsigned(v & mr, rshift, rcount));

out[z++] = STBI\_\_BYTECAST(stbi\_\_shiftsigned(v & mg, gshift, gcount));

out[z++] = STBI\_\_BYTECAST(stbi\_\_shiftsigned(v & mb, bshift, bcount));

a = (ma ? stbi\_\_shiftsigned(v & ma, ashift, acount) : 255);

all\_a |= a;

if (target == 4) out[z++] = STBI\_\_BYTECAST(a);

}

}

stbi\_\_skip(s, pad);

}

}

// if alpha channel is all 0s, replace with all 255s

if (target == 4 && all\_a == 0)

for (i=4\*s->img\_x\*s->img\_y-1; i >= 0; i -= 4)

out[i] = 255;

if (flip\_vertically) {

stbi\_uc t;

for (j=0; j < (int) s->img\_y>>1; ++j) {

stbi\_uc \*p1 = out + j \*s->img\_x\*target;

stbi\_uc \*p2 = out + (s->img\_y-1-j)\*s->img\_x\*target;

for (i=0; i < (int) s->img\_x\*target; ++i) {

t = p1[i], p1[i] = p2[i], p2[i] = t;

}

}

}

if (req\_comp && req\_comp != target) {

out = stbi\_\_convert\_format(out, target, req\_comp, s->img\_x, s->img\_y);

if (out == NULL) return out; // stbi\_\_convert\_format frees input on failure

}

\*x = s->img\_x;

\*y = s->img\_y;

if (comp) \*comp = s->img\_n;

return out;

}

#endif

// Targa Truevision - TGA

// by Jonathan Dummer

#ifndef STBI\_NO\_TGA

// returns STBI\_rgb or whatever, 0 on error

static int stbi\_\_tga\_get\_comp(int bits\_per\_pixel, int is\_grey, int\* is\_rgb16)

{

// only RGB or RGBA (incl. 16bit) or grey allowed

if(is\_rgb16) \*is\_rgb16 = 0;

switch(bits\_per\_pixel) {

case 8: return STBI\_grey;

case 16: if(is\_grey) return STBI\_grey\_alpha;

// else: fall-through

case 15: if(is\_rgb16) \*is\_rgb16 = 1;

return STBI\_rgb;

case 24: // fall-through

case 32: return bits\_per\_pixel/8;

default: return 0;

}

}

static int stbi\_\_tga\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

int tga\_w, tga\_h, tga\_comp, tga\_image\_type, tga\_bits\_per\_pixel, tga\_colormap\_bpp;

int sz, tga\_colormap\_type;

stbi\_\_get8(s); // discard Offset

tga\_colormap\_type = stbi\_\_get8(s); // colormap type

if( tga\_colormap\_type > 1 ) {

stbi\_\_rewind(s);

return 0; // only RGB or indexed allowed

}

tga\_image\_type = stbi\_\_get8(s); // image type

if ( tga\_colormap\_type == 1 ) { // colormapped (paletted) image

if (tga\_image\_type != 1 && tga\_image\_type != 9) {

stbi\_\_rewind(s);

return 0;

}

stbi\_\_skip(s,4); // skip index of first colormap entry and number of entries

sz = stbi\_\_get8(s); // check bits per palette color entry

if ( (sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32) ) {

stbi\_\_rewind(s);

return 0;

}

stbi\_\_skip(s,4); // skip image x and y origin

tga\_colormap\_bpp = sz;

} else { // "normal" image w/o colormap - only RGB or grey allowed, +/- RLE

if ( (tga\_image\_type != 2) && (tga\_image\_type != 3) && (tga\_image\_type != 10) && (tga\_image\_type != 11) ) {

stbi\_\_rewind(s);

return 0; // only RGB or grey allowed, +/- RLE

}

stbi\_\_skip(s,9); // skip colormap specification and image x/y origin

tga\_colormap\_bpp = 0;

}

tga\_w = stbi\_\_get16le(s);

if( tga\_w < 1 ) {

stbi\_\_rewind(s);

return 0; // test width

}

tga\_h = stbi\_\_get16le(s);

if( tga\_h < 1 ) {

stbi\_\_rewind(s);

return 0; // test height

}

tga\_bits\_per\_pixel = stbi\_\_get8(s); // bits per pixel

stbi\_\_get8(s); // ignore alpha bits

if (tga\_colormap\_bpp != 0) {

if((tga\_bits\_per\_pixel != 8) && (tga\_bits\_per\_pixel != 16)) {

// when using a colormap, tga\_bits\_per\_pixel is the size of the indexes

// I don't think anything but 8 or 16bit indexes makes sense

stbi\_\_rewind(s);

return 0;

}

tga\_comp = stbi\_\_tga\_get\_comp(tga\_colormap\_bpp, 0, NULL);

} else {

tga\_comp = stbi\_\_tga\_get\_comp(tga\_bits\_per\_pixel, (tga\_image\_type == 3) || (tga\_image\_type == 11), NULL);

}

if(!tga\_comp) {

stbi\_\_rewind(s);

return 0;

}

if (x) \*x = tga\_w;

if (y) \*y = tga\_h;

if (comp) \*comp = tga\_comp;

return 1; // seems to have passed everything

}

static int stbi\_\_tga\_test(stbi\_\_context \*s)

{

int res = 0;

int sz, tga\_color\_type;

stbi\_\_get8(s); // discard Offset

tga\_color\_type = stbi\_\_get8(s); // color type

if ( tga\_color\_type > 1 ) goto errorEnd; // only RGB or indexed allowed

sz = stbi\_\_get8(s); // image type

if ( tga\_color\_type == 1 ) { // colormapped (paletted) image

if (sz != 1 && sz != 9) goto errorEnd; // colortype 1 demands image type 1 or 9

stbi\_\_skip(s,4); // skip index of first colormap entry and number of entries

sz = stbi\_\_get8(s); // check bits per palette color entry

if ( (sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32) ) goto errorEnd;

stbi\_\_skip(s,4); // skip image x and y origin

} else { // "normal" image w/o colormap

if ( (sz != 2) && (sz != 3) && (sz != 10) && (sz != 11) ) goto errorEnd; // only RGB or grey allowed, +/- RLE

stbi\_\_skip(s,9); // skip colormap specification and image x/y origin

}

if ( stbi\_\_get16le(s) < 1 ) goto errorEnd; // test width

if ( stbi\_\_get16le(s) < 1 ) goto errorEnd; // test height

sz = stbi\_\_get8(s); // bits per pixel

if ( (tga\_color\_type == 1) && (sz != 8) && (sz != 16) ) goto errorEnd; // for colormapped images, bpp is size of an index

if ( (sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32) ) goto errorEnd;

res = 1; // if we got this far, everything's good and we can return 1 instead of 0

errorEnd:

stbi\_\_rewind(s);

return res;

}

// read 16bit value and convert to 24bit RGB

static void stbi\_\_tga\_read\_rgb16(stbi\_\_context \*s, stbi\_uc\* out)

{

stbi\_\_uint16 px = (stbi\_\_uint16)stbi\_\_get16le(s);

stbi\_\_uint16 fiveBitMask = 31;

// we have 3 channels with 5bits each

int r = (px >> 10) & fiveBitMask;

int g = (px >> 5) & fiveBitMask;

int b = px & fiveBitMask;

// Note that this saves the data in RGB(A) order, so it doesn't need to be swapped later

out[0] = (stbi\_uc)((r \* 255)/31);

out[1] = (stbi\_uc)((g \* 255)/31);

out[2] = (stbi\_uc)((b \* 255)/31);

// some people claim that the most significant bit might be used for alpha

// (possibly if an alpha-bit is set in the "image descriptor byte")

// but that only made 16bit test images completely translucent..

// so let's treat all 15 and 16bit TGAs as RGB with no alpha.

}

static void \*stbi\_\_tga\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri)

{

// read in the TGA header stuff

int tga\_offset = stbi\_\_get8(s);

int tga\_indexed = stbi\_\_get8(s);

int tga\_image\_type = stbi\_\_get8(s);

int tga\_is\_RLE = 0;

int tga\_palette\_start = stbi\_\_get16le(s);

int tga\_palette\_len = stbi\_\_get16le(s);

int tga\_palette\_bits = stbi\_\_get8(s);

int tga\_x\_origin = stbi\_\_get16le(s);

int tga\_y\_origin = stbi\_\_get16le(s);

int tga\_width = stbi\_\_get16le(s);

int tga\_height = stbi\_\_get16le(s);

int tga\_bits\_per\_pixel = stbi\_\_get8(s);

int tga\_comp, tga\_rgb16=0;

int tga\_inverted = stbi\_\_get8(s);

// int tga\_alpha\_bits = tga\_inverted & 15; // the 4 lowest bits - unused (useless?)

// image data

unsigned char \*tga\_data;

unsigned char \*tga\_palette = NULL;

int i, j;

unsigned char raw\_data[4] = {0};

int RLE\_count = 0;

int RLE\_repeating = 0;

int read\_next\_pixel = 1;

STBI\_NOTUSED(ri);

// do a tiny bit of precessing

if ( tga\_image\_type >= 8 )

{

tga\_image\_type -= 8;

tga\_is\_RLE = 1;

}

tga\_inverted = 1 - ((tga\_inverted >> 5) & 1);

// If I'm paletted, then I'll use the number of bits from the palette

if ( tga\_indexed ) tga\_comp = stbi\_\_tga\_get\_comp(tga\_palette\_bits, 0, &tga\_rgb16);

else tga\_comp = stbi\_\_tga\_get\_comp(tga\_bits\_per\_pixel, (tga\_image\_type == 3), &tga\_rgb16);

if(!tga\_comp) // shouldn't really happen, stbi\_\_tga\_test() should have ensured basic consistency

return stbi\_\_errpuc("bad format", "Can't find out TGA pixelformat");

// tga info

\*x = tga\_width;

\*y = tga\_height;

if (comp) \*comp = tga\_comp;

if (!stbi\_\_mad3sizes\_valid(tga\_width, tga\_height, tga\_comp, 0))

return stbi\_\_errpuc("too large", "Corrupt TGA");

tga\_data = (unsigned char\*)stbi\_\_malloc\_mad3(tga\_width, tga\_height, tga\_comp, 0);

if (!tga\_data) return stbi\_\_errpuc("outofmem", "Out of memory");

// skip to the data's starting position (offset usually = 0)

stbi\_\_skip(s, tga\_offset );

if ( !tga\_indexed && !tga\_is\_RLE && !tga\_rgb16 ) {

for (i=0; i < tga\_height; ++i) {

int row = tga\_inverted ? tga\_height -i - 1 : i;

stbi\_uc \*tga\_row = tga\_data + row\*tga\_width\*tga\_comp;

stbi\_\_getn(s, tga\_row, tga\_width \* tga\_comp);

}

} else {

// do I need to load a palette?

if ( tga\_indexed)

{

// any data to skip? (offset usually = 0)

stbi\_\_skip(s, tga\_palette\_start );

// load the palette

tga\_palette = (unsigned char\*)stbi\_\_malloc\_mad2(tga\_palette\_len, tga\_comp, 0);

if (!tga\_palette) {

STBI\_FREE(tga\_data);

return stbi\_\_errpuc("outofmem", "Out of memory");

}

if (tga\_rgb16) {

stbi\_uc \*pal\_entry = tga\_palette;

STBI\_ASSERT(tga\_comp == STBI\_rgb);

for (i=0; i < tga\_palette\_len; ++i) {

stbi\_\_tga\_read\_rgb16(s, pal\_entry);

pal\_entry += tga\_comp;

}

} else if (!stbi\_\_getn(s, tga\_palette, tga\_palette\_len \* tga\_comp)) {

STBI\_FREE(tga\_data);

STBI\_FREE(tga\_palette);

return stbi\_\_errpuc("bad palette", "Corrupt TGA");

}

}

// load the data

for (i=0; i < tga\_width \* tga\_height; ++i)

{

// if I'm in RLE mode, do I need to get a RLE stbi\_\_pngchunk?

if ( tga\_is\_RLE )

{

if ( RLE\_count == 0 )

{

// yep, get the next byte as a RLE command

int RLE\_cmd = stbi\_\_get8(s);

RLE\_count = 1 + (RLE\_cmd & 127);

RLE\_repeating = RLE\_cmd >> 7;

read\_next\_pixel = 1;

} else if ( !RLE\_repeating )

{

read\_next\_pixel = 1;

}

} else

{

read\_next\_pixel = 1;

}

// OK, if I need to read a pixel, do it now

if ( read\_next\_pixel )

{

// load however much data we did have

if ( tga\_indexed )

{

// read in index, then perform the lookup

int pal\_idx = (tga\_bits\_per\_pixel == 8) ? stbi\_\_get8(s) : stbi\_\_get16le(s);

if ( pal\_idx >= tga\_palette\_len ) {

// invalid index

pal\_idx = 0;

}

pal\_idx \*= tga\_comp;

for (j = 0; j < tga\_comp; ++j) {

raw\_data[j] = tga\_palette[pal\_idx+j];

}

} else if(tga\_rgb16) {

STBI\_ASSERT(tga\_comp == STBI\_rgb);

stbi\_\_tga\_read\_rgb16(s, raw\_data);

} else {

// read in the data raw

for (j = 0; j < tga\_comp; ++j) {

raw\_data[j] = stbi\_\_get8(s);

}

}

// clear the reading flag for the next pixel

read\_next\_pixel = 0;

} // end of reading a pixel

// copy data

for (j = 0; j < tga\_comp; ++j)

tga\_data[i\*tga\_comp+j] = raw\_data[j];

// in case we're in RLE mode, keep counting down

--RLE\_count;

}

// do I need to invert the image?

if ( tga\_inverted )

{

for (j = 0; j\*2 < tga\_height; ++j)

{

int index1 = j \* tga\_width \* tga\_comp;

int index2 = (tga\_height - 1 - j) \* tga\_width \* tga\_comp;

for (i = tga\_width \* tga\_comp; i > 0; --i)

{

unsigned char temp = tga\_data[index1];

tga\_data[index1] = tga\_data[index2];

tga\_data[index2] = temp;

++index1;

++index2;

}

}

}

// clear my palette, if I had one

if ( tga\_palette != NULL )

{

STBI\_FREE( tga\_palette );

}

}

// swap RGB - if the source data was RGB16, it already is in the right order

if (tga\_comp >= 3 && !tga\_rgb16)

{

unsigned char\* tga\_pixel = tga\_data;

for (i=0; i < tga\_width \* tga\_height; ++i)

{

unsigned char temp = tga\_pixel[0];

tga\_pixel[0] = tga\_pixel[2];

tga\_pixel[2] = temp;

tga\_pixel += tga\_comp;

}

}

// convert to target component count

if (req\_comp && req\_comp != tga\_comp)

tga\_data = stbi\_\_convert\_format(tga\_data, tga\_comp, req\_comp, tga\_width, tga\_height);

// the things I do to get rid of an error message, and yet keep

// Microsoft's C compilers happy... [8^(

tga\_palette\_start = tga\_palette\_len = tga\_palette\_bits =

tga\_x\_origin = tga\_y\_origin = 0;

// OK, done

return tga\_data;

}

#endif

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

// Photoshop PSD loader -- PD by Thatcher Ulrich, integration by Nicolas Schulz, tweaked by STB

#ifndef STBI\_NO\_PSD

static int stbi\_\_psd\_test(stbi\_\_context \*s)

{

int r = (stbi\_\_get32be(s) == 0x38425053);

stbi\_\_rewind(s);

return r;

}

static int stbi\_\_psd\_decode\_rle(stbi\_\_context \*s, stbi\_uc \*p, int pixelCount)

{

int count, nleft, len;

count = 0;

while ((nleft = pixelCount - count) > 0) {

len = stbi\_\_get8(s);

if (len == 128) {

// No-op.

} else if (len < 128) {

// Copy next len+1 bytes literally.

len++;

if (len > nleft) return 0; // corrupt data

count += len;

while (len) {

\*p = stbi\_\_get8(s);

p += 4;

len--;

}

} else if (len > 128) {

stbi\_uc val;

// Next -len+1 bytes in the dest are replicated from next source byte.

// (Interpret len as a negative 8-bit int.)

len = 257 - len;

if (len > nleft) return 0; // corrupt data

val = stbi\_\_get8(s);

count += len;

while (len) {

\*p = val;

p += 4;

len--;

}

}

}

return 1;

}

static void \*stbi\_\_psd\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri, int bpc)

{

int pixelCount;

int channelCount, compression;

int channel, i;

int bitdepth;

int w,h;

stbi\_uc \*out;

STBI\_NOTUSED(ri);

// Check identifier

if (stbi\_\_get32be(s) != 0x38425053) // "8BPS"

return stbi\_\_errpuc("not PSD", "Corrupt PSD image");

// Check file type version.

if (stbi\_\_get16be(s) != 1)

return stbi\_\_errpuc("wrong version", "Unsupported version of PSD image");

// Skip 6 reserved bytes.

stbi\_\_skip(s, 6 );

// Read the number of channels (R, G, B, A, etc).

channelCount = stbi\_\_get16be(s);

if (channelCount < 0 || channelCount > 16)

return stbi\_\_errpuc("wrong channel count", "Unsupported number of channels in PSD image");

// Read the rows and columns of the image.

h = stbi\_\_get32be(s);

w = stbi\_\_get32be(s);

// Make sure the depth is 8 bits.

bitdepth = stbi\_\_get16be(s);

if (bitdepth != 8 && bitdepth != 16)

return stbi\_\_errpuc("unsupported bit depth", "PSD bit depth is not 8 or 16 bit");

// Make sure the color mode is RGB.

// Valid options are:

// 0: Bitmap

// 1: Grayscale

// 2: Indexed color

// 3: RGB color

// 4: CMYK color

// 7: Multichannel

// 8: Duotone

// 9: Lab color

if (stbi\_\_get16be(s) != 3)

return stbi\_\_errpuc("wrong color format", "PSD is not in RGB color format");

// Skip the Mode Data. (It's the palette for indexed color; other info for other modes.)

stbi\_\_skip(s,stbi\_\_get32be(s) );

// Skip the image resources. (resolution, pen tool paths, etc)

stbi\_\_skip(s, stbi\_\_get32be(s) );

// Skip the reserved data.

stbi\_\_skip(s, stbi\_\_get32be(s) );

// Find out if the data is compressed.

// Known values:

// 0: no compression

// 1: RLE compressed

compression = stbi\_\_get16be(s);

if (compression > 1)

return stbi\_\_errpuc("bad compression", "PSD has an unknown compression format");

// Check size

if (!stbi\_\_mad3sizes\_valid(4, w, h, 0))

return stbi\_\_errpuc("too large", "Corrupt PSD");

// Create the destination image.

if (!compression && bitdepth == 16 && bpc == 16) {

out = (stbi\_uc \*) stbi\_\_malloc\_mad3(8, w, h, 0);

ri->bits\_per\_channel = 16;

} else

out = (stbi\_uc \*) stbi\_\_malloc(4 \* w\*h);

if (!out) return stbi\_\_errpuc("outofmem", "Out of memory");

pixelCount = w\*h;

// Initialize the data to zero.

//memset( out, 0, pixelCount \* 4 );

// Finally, the image data.

if (compression) {

// RLE as used by .PSD and .TIFF

// Loop until you get the number of unpacked bytes you are expecting:

// Read the next source byte into n.

// If n is between 0 and 127 inclusive, copy the next n+1 bytes literally.

// Else if n is between -127 and -1 inclusive, copy the next byte -n+1 times.

// Else if n is 128, noop.

// Endloop

// The RLE-compressed data is preceeded by a 2-byte data count for each row in the data,

// which we're going to just skip.

stbi\_\_skip(s, h \* channelCount \* 2 );

// Read the RLE data by channel.

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

stbi\_uc \*p;

p = out+channel;

if (channel >= channelCount) {

// Fill this channel with default data.

for (i = 0; i < pixelCount; i++, p += 4)

\*p = (channel == 3 ? 255 : 0);

} else {

// Read the RLE data.

if (!stbi\_\_psd\_decode\_rle(s, p, pixelCount)) {

STBI\_FREE(out);

return stbi\_\_errpuc("corrupt", "bad RLE data");

}

}

}

} else {

// We're at the raw image data. It's each channel in order (Red, Green, Blue, Alpha, ...)

// where each channel consists of an 8-bit (or 16-bit) value for each pixel in the image.

// Read the data by channel.

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

if (channel >= channelCount) {

// Fill this channel with default data.

if (bitdepth == 16 && bpc == 16) {

stbi\_\_uint16 \*q = ((stbi\_\_uint16 \*) out) + channel;

stbi\_\_uint16 val = channel == 3 ? 65535 : 0;

for (i = 0; i < pixelCount; i++, q += 4)

\*q = val;

} else {

stbi\_uc \*p = out+channel;

stbi\_uc val = channel == 3 ? 255 : 0;

for (i = 0; i < pixelCount; i++, p += 4)

\*p = val;

}

} else {

if (ri->bits\_per\_channel == 16) { // output bpc

stbi\_\_uint16 \*q = ((stbi\_\_uint16 \*) out) + channel;

for (i = 0; i < pixelCount; i++, q += 4)

\*q = (stbi\_\_uint16) stbi\_\_get16be(s);

} else {

stbi\_uc \*p = out+channel;

if (bitdepth == 16) { // input bpc

for (i = 0; i < pixelCount; i++, p += 4)

\*p = (stbi\_uc) (stbi\_\_get16be(s) >> 8);

} else {

for (i = 0; i < pixelCount; i++, p += 4)

\*p = stbi\_\_get8(s);

}

}

}

}

}

// remove weird white matte from PSD

if (channelCount >= 4) {

if (ri->bits\_per\_channel == 16) {

for (i=0; i < w\*h; ++i) {

stbi\_\_uint16 \*pixel = (stbi\_\_uint16 \*) out + 4\*i;

if (pixel[3] != 0 && pixel[3] != 65535) {

float a = pixel[3] / 65535.0f;

float ra = 1.0f / a;

float inv\_a = 65535.0f \* (1 - ra);

pixel[0] = (stbi\_\_uint16) (pixel[0]\*ra + inv\_a);

pixel[1] = (stbi\_\_uint16) (pixel[1]\*ra + inv\_a);

pixel[2] = (stbi\_\_uint16) (pixel[2]\*ra + inv\_a);

}

}

} else {

for (i=0; i < w\*h; ++i) {

unsigned char \*pixel = out + 4\*i;

if (pixel[3] != 0 && pixel[3] != 255) {

float a = pixel[3] / 255.0f;

float ra = 1.0f / a;

float inv\_a = 255.0f \* (1 - ra);

pixel[0] = (unsigned char) (pixel[0]\*ra + inv\_a);

pixel[1] = (unsigned char) (pixel[1]\*ra + inv\_a);

pixel[2] = (unsigned char) (pixel[2]\*ra + inv\_a);

}

}

}

}

// convert to desired output format

if (req\_comp && req\_comp != 4) {

if (ri->bits\_per\_channel == 16)

out = (stbi\_uc \*) stbi\_\_convert\_format16((stbi\_\_uint16 \*) out, 4, req\_comp, w, h);

else

out = stbi\_\_convert\_format(out, 4, req\_comp, w, h);

if (out == NULL) return out; // stbi\_\_convert\_format frees input on failure

}

if (comp) \*comp = 4;

\*y = h;

\*x = w;

return out;

}

#endif

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

// Softimage PIC loader

// by Tom Seddon

//

// See http://softimage.wiki.softimage.com/index.php/INFO:\_PIC\_file\_format

// See http://ozviz.wasp.uwa.edu.au/~pbourke/dataformats/softimagepic/

#ifndef STBI\_NO\_PIC

static int stbi\_\_pic\_is4(stbi\_\_context \*s,const char \*str)

{

int i;

for (i=0; i<4; ++i)

if (stbi\_\_get8(s) != (stbi\_uc)str[i])

return 0;

return 1;

}

static int stbi\_\_pic\_test\_core(stbi\_\_context \*s)

{

int i;

if (!stbi\_\_pic\_is4(s,"\x53\x80\xF6\x34"))

return 0;

for(i=0;i<84;++i)

stbi\_\_get8(s);

if (!stbi\_\_pic\_is4(s,"PICT"))

return 0;

return 1;

}

typedef struct

{

stbi\_uc size,type,channel;

} stbi\_\_pic\_packet;

static stbi\_uc \*stbi\_\_readval(stbi\_\_context \*s, int channel, stbi\_uc \*dest)

{

int mask=0x80, i;

for (i=0; i<4; ++i, mask>>=1) {

if (channel & mask) {

if (stbi\_\_at\_eof(s)) return stbi\_\_errpuc("bad file","PIC file too short");

dest[i]=stbi\_\_get8(s);

}

}

return dest;

}

static void stbi\_\_copyval(int channel,stbi\_uc \*dest,const stbi\_uc \*src)

{

int mask=0x80,i;

for (i=0;i<4; ++i, mask>>=1)

if (channel&mask)

dest[i]=src[i];

}

static stbi\_uc \*stbi\_\_pic\_load\_core(stbi\_\_context \*s,int width,int height,int \*comp, stbi\_uc \*result)

{

int act\_comp=0,num\_packets=0,y,chained;

stbi\_\_pic\_packet packets[10];

// this will (should...) cater for even some bizarre stuff like having data

// for the same channel in multiple packets.

do {

stbi\_\_pic\_packet \*packet;

if (num\_packets==sizeof(packets)/sizeof(packets[0]))

return stbi\_\_errpuc("bad format","too many packets");

packet = &packets[num\_packets++];

chained = stbi\_\_get8(s);

packet->size = stbi\_\_get8(s);

packet->type = stbi\_\_get8(s);

packet->channel = stbi\_\_get8(s);

act\_comp |= packet->channel;

if (stbi\_\_at\_eof(s)) return stbi\_\_errpuc("bad file","file too short (reading packets)");

if (packet->size != 8) return stbi\_\_errpuc("bad format","packet isn't 8bpp");

} while (chained);

\*comp = (act\_comp & 0x10 ? 4 : 3); // has alpha channel?

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

int packet\_idx;

for(packet\_idx=0; packet\_idx < num\_packets; ++packet\_idx) {

stbi\_\_pic\_packet \*packet = &packets[packet\_idx];

stbi\_uc \*dest = result+y\*width\*4;

switch (packet->type) {

default:

return stbi\_\_errpuc("bad format","packet has bad compression type");

case 0: {//uncompressed

int x;

for(x=0;x<width;++x, dest+=4)

if (!stbi\_\_readval(s,packet->channel,dest))

return 0;

break;

}

case 1://Pure RLE

{

int left=width, i;

while (left>0) {

stbi\_uc count,value[4];

count=stbi\_\_get8(s);

if (stbi\_\_at\_eof(s)) return stbi\_\_errpuc("bad file","file too short (pure read count)");

if (count > left)

count = (stbi\_uc) left;

if (!stbi\_\_readval(s,packet->channel,value)) return 0;

for(i=0; i<count; ++i,dest+=4)

stbi\_\_copyval(packet->channel,dest,value);

left -= count;

}

}

break;

case 2: {//Mixed RLE

int left=width;

while (left>0) {

int count = stbi\_\_get8(s), i;

if (stbi\_\_at\_eof(s)) return stbi\_\_errpuc("bad file","file too short (mixed read count)");

if (count >= 128) { // Repeated

stbi\_uc value[4];

if (count==128)

count = stbi\_\_get16be(s);

else

count -= 127;

if (count > left)

return stbi\_\_errpuc("bad file","scanline overrun");

if (!stbi\_\_readval(s,packet->channel,value))

return 0;

for(i=0;i<count;++i, dest += 4)

stbi\_\_copyval(packet->channel,dest,value);

} else { // Raw

++count;

if (count>left) return stbi\_\_errpuc("bad file","scanline overrun");

for(i=0;i<count;++i, dest+=4)

if (!stbi\_\_readval(s,packet->channel,dest))

return 0;

}

left-=count;

}

break;

}

}

}

}

return result;

}

static void \*stbi\_\_pic\_load(stbi\_\_context \*s,int \*px,int \*py,int \*comp,int req\_comp, stbi\_\_result\_info \*ri)

{

stbi\_uc \*result;

int i, x,y, internal\_comp;

STBI\_NOTUSED(ri);

if (!comp) comp = &internal\_comp;

for (i=0; i<92; ++i)

stbi\_\_get8(s);

x = stbi\_\_get16be(s);

y = stbi\_\_get16be(s);

if (stbi\_\_at\_eof(s)) return stbi\_\_errpuc("bad file","file too short (pic header)");

if (!stbi\_\_mad3sizes\_valid(x, y, 4, 0)) return stbi\_\_errpuc("too large", "PIC image too large to decode");

stbi\_\_get32be(s); //skip `ratio'

stbi\_\_get16be(s); //skip `fields'

stbi\_\_get16be(s); //skip `pad'

// intermediate buffer is RGBA

result = (stbi\_uc \*) stbi\_\_malloc\_mad3(x, y, 4, 0);

memset(result, 0xff, x\*y\*4);

if (!stbi\_\_pic\_load\_core(s,x,y,comp, result)) {

STBI\_FREE(result);

result=0;

}

\*px = x;

\*py = y;

if (req\_comp == 0) req\_comp = \*comp;

result=stbi\_\_convert\_format(result,4,req\_comp,x,y);

return result;

}

static int stbi\_\_pic\_test(stbi\_\_context \*s)

{

int r = stbi\_\_pic\_test\_core(s);

stbi\_\_rewind(s);

return r;

}

#endif

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

// GIF loader -- public domain by Jean-Marc Lienher -- simplified/shrunk by stb

#ifndef STBI\_NO\_GIF

typedef struct

{

stbi\_\_int16 prefix;

stbi\_uc first;

stbi\_uc suffix;

} stbi\_\_gif\_lzw;

typedef struct

{

int w,h;

stbi\_uc \*out, \*old\_out; // output buffer (always 4 components)

int flags, bgindex, ratio, transparent, eflags, delay;

stbi\_uc pal[256][4];

stbi\_uc lpal[256][4];

stbi\_\_gif\_lzw codes[4096];

stbi\_uc \*color\_table;

int parse, step;

int lflags;

int start\_x, start\_y;

int max\_x, max\_y;

int cur\_x, cur\_y;

int line\_size;

} stbi\_\_gif;

static int stbi\_\_gif\_test\_raw(stbi\_\_context \*s)

{

int sz;

if (stbi\_\_get8(s) != 'G' || stbi\_\_get8(s) != 'I' || stbi\_\_get8(s) != 'F' || stbi\_\_get8(s) != '8') return 0;

sz = stbi\_\_get8(s);

if (sz != '9' && sz != '7') return 0;

if (stbi\_\_get8(s) != 'a') return 0;

return 1;

}

static int stbi\_\_gif\_test(stbi\_\_context \*s)

{

int r = stbi\_\_gif\_test\_raw(s);

stbi\_\_rewind(s);

return r;

}

static void stbi\_\_gif\_parse\_colortable(stbi\_\_context \*s, stbi\_uc pal[256][4], int num\_entries, int transp)

{

int i;

for (i=0; i < num\_entries; ++i) {

pal[i][2] = stbi\_\_get8(s);

pal[i][1] = stbi\_\_get8(s);

pal[i][0] = stbi\_\_get8(s);

pal[i][3] = transp == i ? 0 : 255;

}

}

static int stbi\_\_gif\_header(stbi\_\_context \*s, stbi\_\_gif \*g, int \*comp, int is\_info)

{

stbi\_uc version;

if (stbi\_\_get8(s) != 'G' || stbi\_\_get8(s) != 'I' || stbi\_\_get8(s) != 'F' || stbi\_\_get8(s) != '8')

return stbi\_\_err("not GIF", "Corrupt GIF");

version = stbi\_\_get8(s);

if (version != '7' && version != '9') return stbi\_\_err("not GIF", "Corrupt GIF");

if (stbi\_\_get8(s) != 'a') return stbi\_\_err("not GIF", "Corrupt GIF");

stbi\_\_g\_failure\_reason = "";

g->w = stbi\_\_get16le(s);

g->h = stbi\_\_get16le(s);

g->flags = stbi\_\_get8(s);

g->bgindex = stbi\_\_get8(s);

g->ratio = stbi\_\_get8(s);

g->transparent = -1;

if (comp != 0) \*comp = 4; // can't actually tell whether it's 3 or 4 until we parse the comments

if (is\_info) return 1;

if (g->flags & 0x80)

stbi\_\_gif\_parse\_colortable(s,g->pal, 2 << (g->flags & 7), -1);

return 1;

}

static int stbi\_\_gif\_info\_raw(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

stbi\_\_gif\* g = (stbi\_\_gif\*) stbi\_\_malloc(sizeof(stbi\_\_gif));

if (!stbi\_\_gif\_header(s, g, comp, 1)) {

STBI\_FREE(g);

stbi\_\_rewind( s );

return 0;

}

if (x) \*x = g->w;

if (y) \*y = g->h;

STBI\_FREE(g);

return 1;

}

static void stbi\_\_out\_gif\_code(stbi\_\_gif \*g, stbi\_\_uint16 code)

{

stbi\_uc \*p, \*c;

// recurse to decode the prefixes, since the linked-list is backwards,

// and working backwards through an interleaved image would be nasty

if (g->codes[code].prefix >= 0)

stbi\_\_out\_gif\_code(g, g->codes[code].prefix);

if (g->cur\_y >= g->max\_y) return;

p = &g->out[g->cur\_x + g->cur\_y];

c = &g->color\_table[g->codes[code].suffix \* 4];

if (c[3] >= 128) {

p[0] = c[2];

p[1] = c[1];

p[2] = c[0];

p[3] = c[3];

}

g->cur\_x += 4;

if (g->cur\_x >= g->max\_x) {

g->cur\_x = g->start\_x;

g->cur\_y += g->step;

while (g->cur\_y >= g->max\_y && g->parse > 0) {

g->step = (1 << g->parse) \* g->line\_size;

g->cur\_y = g->start\_y + (g->step >> 1);

--g->parse;

}

}

}

static stbi\_uc \*stbi\_\_process\_gif\_raster(stbi\_\_context \*s, stbi\_\_gif \*g)

{

stbi\_uc lzw\_cs;

stbi\_\_int32 len, init\_code;

stbi\_\_uint32 first;

stbi\_\_int32 codesize, codemask, avail, oldcode, bits, valid\_bits, clear;

stbi\_\_gif\_lzw \*p;

lzw\_cs = stbi\_\_get8(s);

if (lzw\_cs > 12) return NULL;

clear = 1 << lzw\_cs;

first = 1;

codesize = lzw\_cs + 1;

codemask = (1 << codesize) - 1;

bits = 0;

valid\_bits = 0;

for (init\_code = 0; init\_code < clear; init\_code++) {

g->codes[init\_code].prefix = -1;

g->codes[init\_code].first = (stbi\_uc) init\_code;

g->codes[init\_code].suffix = (stbi\_uc) init\_code;

}

// support no starting clear code

avail = clear+2;

oldcode = -1;

len = 0;

for(;;) {

if (valid\_bits < codesize) {

if (len == 0) {

len = stbi\_\_get8(s); // start new block

if (len == 0)

return g->out;

}

--len;

bits |= (stbi\_\_int32) stbi\_\_get8(s) << valid\_bits;

valid\_bits += 8;

} else {

stbi\_\_int32 code = bits & codemask;

bits >>= codesize;

valid\_bits -= codesize;

// @OPTIMIZE: is there some way we can accelerate the non-clear path?

if (code == clear) { // clear code

codesize = lzw\_cs + 1;

codemask = (1 << codesize) - 1;

avail = clear + 2;

oldcode = -1;

first = 0;

} else if (code == clear + 1) { // end of stream code

stbi\_\_skip(s, len);

while ((len = stbi\_\_get8(s)) > 0)

stbi\_\_skip(s,len);

return g->out;

} else if (code <= avail) {

if (first) return stbi\_\_errpuc("no clear code", "Corrupt GIF");

if (oldcode >= 0) {

p = &g->codes[avail++];

if (avail > 4096) return stbi\_\_errpuc("too many codes", "Corrupt GIF");

p->prefix = (stbi\_\_int16) oldcode;

p->first = g->codes[oldcode].first;

p->suffix = (code == avail) ? p->first : g->codes[code].first;

} else if (code == avail)

return stbi\_\_errpuc("illegal code in raster", "Corrupt GIF");

stbi\_\_out\_gif\_code(g, (stbi\_\_uint16) code);

if ((avail & codemask) == 0 && avail <= 0x0FFF) {

codesize++;

codemask = (1 << codesize) - 1;

}

oldcode = code;

} else {

return stbi\_\_errpuc("illegal code in raster", "Corrupt GIF");

}

}

}

}

static void stbi\_\_fill\_gif\_background(stbi\_\_gif \*g, int x0, int y0, int x1, int y1)

{

int x, y;

stbi\_uc \*c = g->pal[g->bgindex];

for (y = y0; y < y1; y += 4 \* g->w) {

for (x = x0; x < x1; x += 4) {

stbi\_uc \*p = &g->out[y + x];

p[0] = c[2];

p[1] = c[1];

p[2] = c[0];

p[3] = 0;

}

}

}

// this function is designed to support animated gifs, although stb\_image doesn't support it

static stbi\_uc \*stbi\_\_gif\_load\_next(stbi\_\_context \*s, stbi\_\_gif \*g, int \*comp, int req\_comp)

{

int i;

stbi\_uc \*prev\_out = 0;

if (g->out == 0 && !stbi\_\_gif\_header(s, g, comp,0))

return 0; // stbi\_\_g\_failure\_reason set by stbi\_\_gif\_header

if (!stbi\_\_mad3sizes\_valid(g->w, g->h, 4, 0))

return stbi\_\_errpuc("too large", "GIF too large");

prev\_out = g->out;

g->out = (stbi\_uc \*) stbi\_\_malloc\_mad3(4, g->w, g->h, 0);

if (g->out == 0) return stbi\_\_errpuc("outofmem", "Out of memory");

switch ((g->eflags & 0x1C) >> 2) {

case 0: // unspecified (also always used on 1st frame)

stbi\_\_fill\_gif\_background(g, 0, 0, 4 \* g->w, 4 \* g->w \* g->h);

break;

case 1: // do not dispose

if (prev\_out) memcpy(g->out, prev\_out, 4 \* g->w \* g->h);

g->old\_out = prev\_out;

break;

case 2: // dispose to background

if (prev\_out) memcpy(g->out, prev\_out, 4 \* g->w \* g->h);

stbi\_\_fill\_gif\_background(g, g->start\_x, g->start\_y, g->max\_x, g->max\_y);

break;

case 3: // dispose to previous

if (g->old\_out) {

for (i = g->start\_y; i < g->max\_y; i += 4 \* g->w)

memcpy(&g->out[i + g->start\_x], &g->old\_out[i + g->start\_x], g->max\_x - g->start\_x);

}

break;

}

for (;;) {

switch (stbi\_\_get8(s)) {

case 0x2C: /\* Image Descriptor \*/

{

int prev\_trans = -1;

stbi\_\_int32 x, y, w, h;

stbi\_uc \*o;

x = stbi\_\_get16le(s);

y = stbi\_\_get16le(s);

w = stbi\_\_get16le(s);

h = stbi\_\_get16le(s);

if (((x + w) > (g->w)) || ((y + h) > (g->h)))

return stbi\_\_errpuc("bad Image Descriptor", "Corrupt GIF");

g->line\_size = g->w \* 4;

g->start\_x = x \* 4;

g->start\_y = y \* g->line\_size;

g->max\_x = g->start\_x + w \* 4;

g->max\_y = g->start\_y + h \* g->line\_size;

g->cur\_x = g->start\_x;

g->cur\_y = g->start\_y;

g->lflags = stbi\_\_get8(s);

if (g->lflags & 0x40) {

g->step = 8 \* g->line\_size; // first interlaced spacing

g->parse = 3;

} else {

g->step = g->line\_size;

g->parse = 0;

}

if (g->lflags & 0x80) {

stbi\_\_gif\_parse\_colortable(s,g->lpal, 2 << (g->lflags & 7), g->eflags & 0x01 ? g->transparent : -1);

g->color\_table = (stbi\_uc \*) g->lpal;

} else if (g->flags & 0x80) {

if (g->transparent >= 0 && (g->eflags & 0x01)) {

prev\_trans = g->pal[g->transparent][3];

g->pal[g->transparent][3] = 0;

}

g->color\_table = (stbi\_uc \*) g->pal;

} else

return stbi\_\_errpuc("missing color table", "Corrupt GIF");

o = stbi\_\_process\_gif\_raster(s, g);

if (o == NULL) return NULL;

if (prev\_trans != -1)

g->pal[g->transparent][3] = (stbi\_uc) prev\_trans;

return o;

}

case 0x21: // Comment Extension.

{

int len;

if (stbi\_\_get8(s) == 0xF9) { // Graphic Control Extension.

len = stbi\_\_get8(s);

if (len == 4) {

g->eflags = stbi\_\_get8(s);

g->delay = stbi\_\_get16le(s);

g->transparent = stbi\_\_get8(s);

} else {

stbi\_\_skip(s, len);

break;

}

}

while ((len = stbi\_\_get8(s)) != 0)

stbi\_\_skip(s, len);

break;

}

case 0x3B: // gif stream termination code

return (stbi\_uc \*) s; // using '1' causes warning on some compilers

default:

return stbi\_\_errpuc("unknown code", "Corrupt GIF");

}

}

STBI\_NOTUSED(req\_comp);

}

static void \*stbi\_\_gif\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri)

{

stbi\_uc \*u = 0;

stbi\_\_gif\* g = (stbi\_\_gif\*) stbi\_\_malloc(sizeof(stbi\_\_gif));

memset(g, 0, sizeof(\*g));

STBI\_NOTUSED(ri);

u = stbi\_\_gif\_load\_next(s, g, comp, req\_comp);

if (u == (stbi\_uc \*) s) u = 0; // end of animated gif marker

if (u) {

\*x = g->w;

\*y = g->h;

if (req\_comp && req\_comp != 4)

u = stbi\_\_convert\_format(u, 4, req\_comp, g->w, g->h);

}

else if (g->out)

STBI\_FREE(g->out);

STBI\_FREE(g);

return u;

}

static int stbi\_\_gif\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

return stbi\_\_gif\_info\_raw(s,x,y,comp);

}

#endif

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

// Radiance RGBE HDR loader

// originally by Nicolas Schulz

#ifndef STBI\_NO\_HDR

static int stbi\_\_hdr\_test\_core(stbi\_\_context \*s, const char \*signature)

{

int i;

for (i=0; signature[i]; ++i)

if (stbi\_\_get8(s) != signature[i])

return 0;

stbi\_\_rewind(s);

return 1;

}

static int stbi\_\_hdr\_test(stbi\_\_context\* s)

{

int r = stbi\_\_hdr\_test\_core(s, "#?RADIANCE\n");

stbi\_\_rewind(s);

if(!r) {

r = stbi\_\_hdr\_test\_core(s, "#?RGBE\n");

stbi\_\_rewind(s);

}

return r;

}

#define STBI\_\_HDR\_BUFLEN 1024

static char \*stbi\_\_hdr\_gettoken(stbi\_\_context \*z, char \*buffer)

{

int len=0;

char c = '\0';

c = (char) stbi\_\_get8(z);

while (!stbi\_\_at\_eof(z) && c != '\n') {

buffer[len++] = c;

if (len == STBI\_\_HDR\_BUFLEN-1) {

// flush to end of line

while (!stbi\_\_at\_eof(z) && stbi\_\_get8(z) != '\n')

;

break;

}

c = (char) stbi\_\_get8(z);

}

buffer[len] = 0;

return buffer;

}

static void stbi\_\_hdr\_convert(float \*output, stbi\_uc \*input, int req\_comp)

{

if ( input[3] != 0 ) {

float f1;

// Exponent

f1 = (float) ldexp(1.0f, input[3] - (int)(128 + 8));

if (req\_comp <= 2)

output[0] = (input[0] + input[1] + input[2]) \* f1 / 3;

else {

output[0] = input[0] \* f1;

output[1] = input[1] \* f1;

output[2] = input[2] \* f1;

}

if (req\_comp == 2) output[1] = 1;

if (req\_comp == 4) output[3] = 1;

} else {

switch (req\_comp) {

case 4: output[3] = 1; /\* fallthrough \*/

case 3: output[0] = output[1] = output[2] = 0;

break;

case 2: output[1] = 1; /\* fallthrough \*/

case 1: output[0] = 0;

break;

}

}

}

static float \*stbi\_\_hdr\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri)

{

char buffer[STBI\_\_HDR\_BUFLEN];

char \*token;

int valid = 0;

int width, height;

stbi\_uc \*scanline;

float \*hdr\_data;

int len;

unsigned char count, value;

int i, j, k, c1,c2, z;

const char \*headerToken;

STBI\_NOTUSED(ri);

// Check identifier

headerToken = stbi\_\_hdr\_gettoken(s,buffer);

if (strcmp(headerToken, "#?RADIANCE") != 0 && strcmp(headerToken, "#?RGBE") != 0)

return stbi\_\_errpf("not HDR", "Corrupt HDR image");

// Parse header

for(;;) {

token = stbi\_\_hdr\_gettoken(s,buffer);

if (token[0] == 0) break;

if (strcmp(token, "FORMAT=32-bit\_rle\_rgbe") == 0) valid = 1;

}

if (!valid) return stbi\_\_errpf("unsupported format", "Unsupported HDR format");

// Parse width and height

// can't use sscanf() if we're not using stdio!

token = stbi\_\_hdr\_gettoken(s,buffer);

if (strncmp(token, "-Y ", 3)) return stbi\_\_errpf("unsupported data layout", "Unsupported HDR format");

token += 3;

height = (int) strtol(token, &token, 10);

while (\*token == ' ') ++token;

if (strncmp(token, "+X ", 3)) return stbi\_\_errpf("unsupported data layout", "Unsupported HDR format");

token += 3;

width = (int) strtol(token, NULL, 10);

\*x = width;

\*y = height;

if (comp) \*comp = 3;

if (req\_comp == 0) req\_comp = 3;

if (!stbi\_\_mad4sizes\_valid(width, height, req\_comp, sizeof(float), 0))

return stbi\_\_errpf("too large", "HDR image is too large");

// Read data

hdr\_data = (float \*) stbi\_\_malloc\_mad4(width, height, req\_comp, sizeof(float), 0);

if (!hdr\_data)

return stbi\_\_errpf("outofmem", "Out of memory");

// Load image data

// image data is stored as some number of sca

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

// Read flat data

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

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

stbi\_uc rgbe[4];

main\_decode\_loop:

stbi\_\_getn(s, rgbe, 4);

stbi\_\_hdr\_convert(hdr\_data + j \* width \* req\_comp + i \* req\_comp, rgbe, req\_comp);

}

}

} else {

// Read RLE-encoded data

scanline = NULL;

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

c1 = stbi\_\_get8(s);

c2 = stbi\_\_get8(s);

len = stbi\_\_get8(s);

if (c1 != 2 || c2 != 2 || (len & 0x80)) {

// not run-length encoded, so we have to actually use THIS data as a decoded

// pixel (note this can't be a valid pixel--one of RGB must be >= 128)

stbi\_uc rgbe[4];

rgbe[0] = (stbi\_uc) c1;

rgbe[1] = (stbi\_uc) c2;

rgbe[2] = (stbi\_uc) len;

rgbe[3] = (stbi\_uc) stbi\_\_get8(s);

stbi\_\_hdr\_convert(hdr\_data, rgbe, req\_comp);

i = 1;

j = 0;

STBI\_FREE(scanline);

goto main\_decode\_loop; // yes, this makes no sense

}

len <<= 8;

len |= stbi\_\_get8(s);

if (len != width) { STBI\_FREE(hdr\_data); STBI\_FREE(scanline); return stbi\_\_errpf("invalid decoded scanline length", "corrupt HDR"); }

if (scanline == NULL) {

scanline = (stbi\_uc \*) stbi\_\_malloc\_mad2(width, 4, 0);

if (!scanline) {

STBI\_FREE(hdr\_data);

return stbi\_\_errpf("outofmem", "Out of memory");

}

}

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

int nleft;

i = 0;

while ((nleft = width - i) > 0) {

count = stbi\_\_get8(s);

if (count > 128) {

// Run

value = stbi\_\_get8(s);

count -= 128;

if (count > nleft) { STBI\_FREE(hdr\_data); STBI\_FREE(scanline); return stbi\_\_errpf("corrupt", "bad RLE data in HDR"); }

for (z = 0; z < count; ++z)

scanline[i++ \* 4 + k] = value;

} else {

// Dump

if (count > nleft) { STBI\_FREE(hdr\_data); STBI\_FREE(scanline); return stbi\_\_errpf("corrupt", "bad RLE data in HDR"); }

for (z = 0; z < count; ++z)

scanline[i++ \* 4 + k] = stbi\_\_get8(s);

}

}

}

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

stbi\_\_hdr\_convert(hdr\_data+(j\*width + i)\*req\_comp, scanline + i\*4, req\_comp);

}

if (scanline)

STBI\_FREE(scanline);

}

return hdr\_data;

}

static int stbi\_\_hdr\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

char buffer[STBI\_\_HDR\_BUFLEN];

char \*token;

int valid = 0;

int dummy;

if (!x) x = &dummy;

if (!y) y = &dummy;

if (!comp) comp = &dummy;

if (stbi\_\_hdr\_test(s) == 0) {

stbi\_\_rewind( s );

return 0;

}

for(;;) {

token = stbi\_\_hdr\_gettoken(s,buffer);

if (token[0] == 0) break;

if (strcmp(token, "FORMAT=32-bit\_rle\_rgbe") == 0) valid = 1;

}

if (!valid) {

stbi\_\_rewind( s );

return 0;

}

token = stbi\_\_hdr\_gettoken(s,buffer);

if (strncmp(token, "-Y ", 3)) {

stbi\_\_rewind( s );

return 0;

}

token += 3;

\*y = (int) strtol(token, &token, 10);

while (\*token == ' ') ++token;

if (strncmp(token, "+X ", 3)) {

stbi\_\_rewind( s );

return 0;

}

token += 3;

\*x = (int) strtol(token, NULL, 10);

\*comp = 3;

return 1;

}

#endif // STBI\_NO\_HDR

#ifndef STBI\_NO\_BMP

static int stbi\_\_bmp\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

void \*p;

stbi\_\_bmp\_data info;

info.all\_a = 255;

p = stbi\_\_bmp\_parse\_header(s, &info);

stbi\_\_rewind( s );

if (p == NULL)

return 0;

if (x) \*x = s->img\_x;

if (y) \*y = s->img\_y;

if (comp) \*comp = info.ma ? 4 : 3;

return 1;

}

#endif

#ifndef STBI\_NO\_PSD

static int stbi\_\_psd\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

int channelCount, dummy;

if (!x) x = &dummy;

if (!y) y = &dummy;

if (!comp) comp = &dummy;

if (stbi\_\_get32be(s) != 0x38425053) {

stbi\_\_rewind( s );

return 0;

}

if (stbi\_\_get16be(s) != 1) {

stbi\_\_rewind( s );

return 0;

}

stbi\_\_skip(s, 6);

channelCount = stbi\_\_get16be(s);

if (channelCount < 0 || channelCount > 16) {

stbi\_\_rewind( s );

return 0;

}

\*y = stbi\_\_get32be(s);

\*x = stbi\_\_get32be(s);

if (stbi\_\_get16be(s) != 8) {

stbi\_\_rewind( s );

return 0;

}

if (stbi\_\_get16be(s) != 3) {

stbi\_\_rewind( s );

return 0;

}

\*comp = 4;

return 1;

}

#endif

#ifndef STBI\_NO\_PIC

static int stbi\_\_pic\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

int act\_comp=0,num\_packets=0,chained,dummy;

stbi\_\_pic\_packet packets[10];

if (!x) x = &dummy;

if (!y) y = &dummy;

if (!comp) comp = &dummy;

if (!stbi\_\_pic\_is4(s,"\x53\x80\xF6\x34")) {

stbi\_\_rewind(s);

return 0;

}

stbi\_\_skip(s, 88);

\*x = stbi\_\_get16be(s);

\*y = stbi\_\_get16be(s);

if (stbi\_\_at\_eof(s)) {

stbi\_\_rewind( s);

return 0;

}

if ( (\*x) != 0 && (1 << 28) / (\*x) < (\*y)) {

stbi\_\_rewind( s );

return 0;

}

stbi\_\_skip(s, 8);

do {

stbi\_\_pic\_packet \*packet;

if (num\_packets==sizeof(packets)/sizeof(packets[0]))

return 0;

packet = &packets[num\_packets++];

chained = stbi\_\_get8(s);

packet->size = stbi\_\_get8(s);

packet->type = stbi\_\_get8(s);

packet->channel = stbi\_\_get8(s);

act\_comp |= packet->channel;

if (stbi\_\_at\_eof(s)) {

stbi\_\_rewind( s );

return 0;

}

if (packet->size != 8) {

stbi\_\_rewind( s );

return 0;

}

} while (chained);

\*comp = (act\_comp & 0x10 ? 4 : 3);

return 1;

}

#endif

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

// Portable Gray Map and Portable Pixel Map loader

// by Ken Miller

//

// PGM: http://netpbm.sourceforge.net/doc/pgm.html

// PPM: http://netpbm.sourceforge.net/doc/ppm.html

//

// Known limitations:

// Does not support comments in the header section

// Does not support ASCII image data (formats P2 and P3)

// Does not support 16-bit-per-channel

#ifndef STBI\_NO\_PNM

static int stbi\_\_pnm\_test(stbi\_\_context \*s)

{

char p, t;

p = (char) stbi\_\_get8(s);

t = (char) stbi\_\_get8(s);

if (p != 'P' || (t != '5' && t != '6')) {

stbi\_\_rewind( s );

return 0;

}

return 1;

}

static void \*stbi\_\_pnm\_load(stbi\_\_context \*s, int \*x, int \*y, int \*comp, int req\_comp, stbi\_\_result\_info \*ri)

{

stbi\_uc \*out;

STBI\_NOTUSED(ri);

if (!stbi\_\_pnm\_info(s, (int \*)&s->img\_x, (int \*)&s->img\_y, (int \*)&s->img\_n))

return 0;

\*x = s->img\_x;

\*y = s->img\_y;

if (comp) \*comp = s->img\_n;

if (!stbi\_\_mad3sizes\_valid(s->img\_n, s->img\_x, s->img\_y, 0))

return stbi\_\_errpuc("too large", "PNM too large");

out = (stbi\_uc \*) stbi\_\_malloc\_mad3(s->img\_n, s->img\_x, s->img\_y, 0);

if (!out) return stbi\_\_errpuc("outofmem", "Out of memory");

stbi\_\_getn(s, out, s->img\_n \* s->img\_x \* s->img\_y);

if (req\_comp && req\_comp != s->img\_n) {

out = stbi\_\_convert\_format(out, s->img\_n, req\_comp, s->img\_x, s->img\_y);

if (out == NULL) return out; // stbi\_\_convert\_format frees input on failure

}

return out;

}

static int stbi\_\_pnm\_isspace(char c)

{

return c == ' ' || c == '\t' || c == '\n' || c == '\v' || c == '\f' || c == '\r';

}

static void stbi\_\_pnm\_skip\_whitespace(stbi\_\_context \*s, char \*c)

{

for (;;) {

while (!stbi\_\_at\_eof(s) && stbi\_\_pnm\_isspace(\*c))

\*c = (char) stbi\_\_get8(s);

if (stbi\_\_at\_eof(s) || \*c != '#')

break;

while (!stbi\_\_at\_eof(s) && \*c != '\n' && \*c != '\r' )

\*c = (char) stbi\_\_get8(s);

}

}

static int stbi\_\_pnm\_isdigit(char c)

{

return c >= '0' && c <= '9';

}

static int stbi\_\_pnm\_getinteger(stbi\_\_context \*s, char \*c)

{

int value = 0;

while (!stbi\_\_at\_eof(s) && stbi\_\_pnm\_isdigit(\*c)) {

value = value\*10 + (\*c - '0');

\*c = (char) stbi\_\_get8(s);

}

return value;

}

static int stbi\_\_pnm\_info(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

int maxv, dummy;

char c, p, t;

if (!x) x = &dummy;

if (!y) y = &dummy;

if (!comp) comp = &dummy;

stbi\_\_rewind(s);

// Get identifier

p = (char) stbi\_\_get8(s);

t = (char) stbi\_\_get8(s);

if (p != 'P' || (t != '5' && t != '6')) {

stbi\_\_rewind(s);

return 0;

}

\*comp = (t == '6') ? 3 : 1; // '5' is 1-component .pgm; '6' is 3-component .ppm

c = (char) stbi\_\_get8(s);

stbi\_\_pnm\_skip\_whitespace(s, &c);

\*x = stbi\_\_pnm\_getinteger(s, &c); // read width

stbi\_\_pnm\_skip\_whitespace(s, &c);

\*y = stbi\_\_pnm\_getinteger(s, &c); // read height

stbi\_\_pnm\_skip\_whitespace(s, &c);

maxv = stbi\_\_pnm\_getinteger(s, &c); // read max value

if (maxv > 255)

return stbi\_\_err("max value > 255", "PPM image not 8-bit");

else

return 1;

}

#endif

static int stbi\_\_info\_main(stbi\_\_context \*s, int \*x, int \*y, int \*comp)

{

#ifndef STBI\_NO\_JPEG

if (stbi\_\_jpeg\_info(s, x, y, comp)) return 1;

#endif

#ifndef STBI\_NO\_PNG

if (stbi\_\_png\_info(s, x, y, comp)) return 1;

#endif

#ifndef STBI\_NO\_GIF

if (stbi\_\_gif\_info(s, x, y, comp)) return 1;

#endif

#ifndef STBI\_NO\_BMP

if (stbi\_\_bmp\_info(s, x, y, comp)) return 1;

#endif

#ifndef STBI\_NO\_PSD

if (stbi\_\_psd\_info(s, x, y, comp)) return 1;

#endif

#ifndef STBI\_NO\_PIC

if (stbi\_\_pic\_info(s, x, y, comp)) return 1;

#endif

#ifndef STBI\_NO\_PNM

if (stbi\_\_pnm\_info(s, x, y, comp)) return 1;

#endif

#ifndef STBI\_NO\_HDR

if (stbi\_\_hdr\_info(s, x, y, comp)) return 1;

#endif

// test tga last because it's a crappy test!

#ifndef STBI\_NO\_TGA

if (stbi\_\_tga\_info(s, x, y, comp))

return 1;

#endif

return stbi\_\_err("unknown image type", "Image not of any known type, or corrupt");

}

#ifndef STBI\_NO\_STDIO

STBIDEF int stbi\_info(char const \*filename, int \*x, int \*y, int \*comp)

{

FILE \*f = stbi\_\_fopen(filename, "rb");

int result;

if (!f) return stbi\_\_err("can't fopen", "Unable to open file");

result = stbi\_info\_from\_file(f, x, y, comp);

fclose(f);

return result;

}

STBIDEF int stbi\_info\_from\_file(FILE \*f, int \*x, int \*y, int \*comp)

{

int r;

stbi\_\_context s;

long pos = ftell(f);

stbi\_\_start\_file(&s, f);

r = stbi\_\_info\_main(&s,x,y,comp);

fseek(f,pos,SEEK\_SET);

return r;

}

#endif // !STBI\_NO\_STDIO

STBIDEF int stbi\_info\_from\_memory(stbi\_uc const \*buffer, int len, int \*x, int \*y, int \*comp)

{

stbi\_\_context s;

stbi\_\_start\_mem(&s,buffer,len);

return stbi\_\_info\_main(&s,x,y,comp);

}

STBIDEF int stbi\_info\_from\_callbacks(stbi\_io\_callbacks const \*c, void \*user, int \*x, int \*y, int \*comp)

{

stbi\_\_context s;

stbi\_\_start\_callbacks(&s, (stbi\_io\_callbacks \*) c, user);

return stbi\_\_info\_main(&s,x,y,comp);

}

#endif // STB\_IMAGE\_IMPLEMENTATION

/\*

revision history:

2.16 (2017-07-23) all functions have 16-bit variants;

STBI\_NO\_STDIO works again;

compilation fixes;

fix rounding in unpremultiply;

optimize vertical flip;

disable raw\_len validation;

documentation fixes

2.15 (2017-03-18) fix png-1,2,4 bug; now all Imagenet JPGs decode;

warning fixes; disable run-time SSE detection on gcc;

uniform handling of optional "return" values;

thread-safe initialization of zlib tables

2.14 (2017-03-03) remove deprecated STBI\_JPEG\_OLD; fixes for Imagenet JPGs

2.13 (2016-11-29) add 16-bit API, only supported for PNG right now

2.12 (2016-04-02) fix typo in 2.11 PSD fix that caused crashes

2.11 (2016-04-02) allocate large structures on the stack

remove white matting for transparent PSD

fix reported channel count for PNG & BMP

re-enable SSE2 in non-gcc 64-bit

support RGB-formatted JPEG

read 16-bit PNGs (only as 8-bit)

2.10 (2016-01-22) avoid warning introduced in 2.09 by STBI\_REALLOC\_SIZED

2.09 (2016-01-16) allow comments in PNM files

16-bit-per-pixel TGA (not bit-per-component)

info() for TGA could break due to .hdr handling

info() for BMP to shares code instead of sloppy parse

can use STBI\_REALLOC\_SIZED if allocator doesn't support realloc

code cleanup

2.08 (2015-09-13) fix to 2.07 cleanup, reading RGB PSD as RGBA

2.07 (2015-09-13) fix compiler warnings

partial animated GIF support

limited 16-bpc PSD support

#ifdef unused functions

bug with < 92 byte PIC,PNM,HDR,TGA

2.06 (2015-04-19) fix bug where PSD returns wrong '\*comp' value

2.05 (2015-04-19) fix bug in progressive JPEG handling, fix warning

2.04 (2015-04-15) try to re-enable SIMD on MinGW 64-bit

2.03 (2015-04-12) extra corruption checking (mmozeiko)

stbi\_set\_flip\_vertically\_on\_load (nguillemot)

fix NEON support; fix mingw support

2.02 (2015-01-19) fix incorrect assert, fix warning

2.01 (2015-01-17) fix various warnings; suppress SIMD on gcc 32-bit without -msse2

2.00b (2014-12-25) fix STBI\_MALLOC in progressive JPEG

2.00 (2014-12-25) optimize JPG, including x86 SSE2 & NEON SIMD (ryg)

progressive JPEG (stb)

PGM/PPM support (Ken Miller)

STBI\_MALLOC,STBI\_REALLOC,STBI\_FREE

GIF bugfix -- seemingly never worked

STBI\_NO\_\*, STBI\_ONLY\_\*

1.48 (2014-12-14) fix incorrectly-named assert()

1.47 (2014-12-14) 1/2/4-bit PNG support, both direct and paletted (Omar Cornut & stb)

optimize PNG (ryg)

fix bug in interlaced PNG with user-specified channel count (stb)

1.46 (2014-08-26)

fix broken tRNS chunk (colorkey-style transparency) in non-paletted PNG

1.45 (2014-08-16)

fix MSVC-ARM internal compiler error by wrapping malloc

1.44 (2014-08-07)

various warning fixes from Ronny Chevalier

1.43 (2014-07-15)

fix MSVC-only compiler problem in code changed in 1.42

1.42 (2014-07-09)

don't define \_CRT\_SECURE\_NO\_WARNINGS (affects user code)

fixes to stbi\_\_cleanup\_jpeg path

added STBI\_ASSERT to avoid requiring assert.h

1.41 (2014-06-25)

fix search&replace from 1.36 that messed up comments/error messages

1.40 (2014-06-22)

fix gcc struct-initialization warning

1.39 (2014-06-15)

fix to TGA optimization when req\_comp != number of components in TGA;

fix to GIF loading because BMP wasn't rewinding (whoops, no GIFs in my test suite)

add support for BMP version 5 (more ignored fields)

1.38 (2014-06-06)

suppress MSVC warnings on integer casts truncating values

fix accidental rename of 'skip' field of I/O

1.37 (2014-06-04)

remove duplicate typedef

1.36 (2014-06-03)

convert to header file single-file library

if de-iphone isn't set, load iphone images color-swapped instead of returning NULL

1.35 (2014-05-27)

various warnings

fix broken STBI\_SIMD path

fix bug where stbi\_load\_from\_file no longer left file pointer in correct place

fix broken non-easy path for 32-bit BMP (possibly never used)

TGA optimization by Arseny Kapoulkine

1.34 (unknown)

use STBI\_NOTUSED in stbi\_\_resample\_row\_generic(), fix one more leak in tga failure case

1.33 (2011-07-14)

make stbi\_is\_hdr work in STBI\_NO\_HDR (as specified), minor compiler-friendly improvements

1.32 (2011-07-13)

support for "info" function for all supported filetypes (SpartanJ)

1.31 (2011-06-20)

a few more leak fixes, bug in PNG handling (SpartanJ)

1.30 (2011-06-11)

added ability to load files via callbacks to accomidate custom input streams (Ben Wenger)

removed deprecated format-specific test/load functions

removed support for installable file formats (stbi\_loader) -- would have been broken for IO callbacks anyway

error cases in bmp and tga give messages and don't leak (Raymond Barbiero, grisha)

fix inefficiency in decoding 32-bit BMP (David Woo)

1.29 (2010-08-16)

various warning fixes from Aurelien Pocheville

1.28 (2010-08-01)

fix bug in GIF palette transparency (SpartanJ)

1.27 (2010-08-01)

cast-to-stbi\_uc to fix warnings

1.26 (2010-07-24)

fix bug in file buffering for PNG reported by SpartanJ

1.25 (2010-07-17)

refix trans\_data warning (Won Chun)

1.24 (2010-07-12)

perf improvements reading from files on platforms with lock-heavy fgetc()

minor perf improvements for jpeg

deprecated type-specific functions so we'll get feedback if they're needed

attempt to fix trans\_data warning (Won Chun)

1.23 fixed bug in iPhone support

1.22 (2010-07-10)

removed image \*writing\* support

stbi\_info support from Jetro Lauha

GIF support from Jean-Marc Lienher

iPhone PNG-extensions from James Brown

warning-fixes from Nicolas Schulz and Janez Zemva (i.stbi\_\_err. Janez (U+017D)emva)

1.21 fix use of 'stbi\_uc' in header (reported by jon blow)

1.20 added support for Softimage PIC, by Tom Seddon

1.19 bug in interlaced PNG corruption check (found by ryg)

1.18 (2008-08-02)

fix a threading bug (local mutable static)

1.17 support interlaced PNG

1.16 major bugfix - stbi\_\_convert\_format converted one too many pixels

1.15 initialize some fields for thread safety

1.14 fix threadsafe conversion bug

header-file-only version (#define STBI\_HEADER\_FILE\_ONLY before including)

1.13 threadsafe

1.12 const qualifiers in the API

1.11 Support installable IDCT, colorspace conversion routines

1.10 Fixes for 64-bit (don't use "unsigned long")

optimized upsampling by Fabian "ryg" Giesen

1.09 Fix format-conversion for PSD code (bad global variables!)

1.08 Thatcher Ulrich's PSD code integrated by Nicolas Schulz

1.07 attempt to fix C++ warning/errors again

1.06 attempt to fix C++ warning/errors again

1.05 fix TGA loading to return correct \*comp and use good luminance calc

1.04 default float alpha is 1, not 255; use 'void \*' for stbi\_image\_free

1.03 bugfixes to STBI\_NO\_STDIO, STBI\_NO\_HDR

1.02 support for (subset of) HDR files, float interface for preferred access to them

1.01 fix bug: possible bug in handling right-side up bmps... not sure

fix bug: the stbi\_\_bmp\_load() and stbi\_\_tga\_load() functions didn't work at all

1.00 interface to zlib that skips zlib header

0.99 correct handling of alpha in palette

0.98 TGA loader by lonesock; dynamically add loaders (untested)

0.97 jpeg errors on too large a file; also catch another malloc failure

0.96 fix detection of invalid v value - particleman@mollyrocket forum

0.95 during header scan, seek to markers in case of padding

0.94 STBI\_NO\_STDIO to disable stdio usage; rename all #defines the same

0.93 handle jpegtran output; verbose errors

0.92 read 4,8,16,24,32-bit BMP files of several formats

0.91 output 24-bit Windows 3.0 BMP files

0.90 fix a few more warnings; bump version number to approach 1.0

0.61 bugfixes due to Marc LeBlanc, Christopher Lloyd

0.60 fix compiling as c++

0.59 fix warnings: merge Dave Moore's -Wall fixes

0.58 fix bug: zlib uncompressed mode len/nlen was wrong endian

0.57 fix bug: jpg last huffman symbol before marker was >9 bits but less than 16 available

0.56 fix bug: zlib uncompressed mode len vs. nlen

0.55 fix bug: restart\_interval not initialized to 0

0.54 allow NULL for 'int \*comp'

0.53 fix bug in png 3->4; speedup png decoding

0.52 png handles req\_comp=3,4 directly; minor cleanup; jpeg comments

0.51 obey req\_comp requests, 1-component jpegs return as 1-component,

on 'test' only check type, not whether we support this variant

0.50 (2006-11-19)

first released version

\*/

/\*

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

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.

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

\*/