Justin Borczuk
Jacob Gold
Maxwell Hu
Shiv Sakhuja
Marco Starger

Programming Languages and Translators, Fall 2017

PIXL Final Report

(Trump photos are ironic)

Contents

Chapter 1

Introduction

PIXL is a language purposed for image processing and design applications. PIXL allows users to take advantage of the pixel as a primitive type, and offers built in operators and library functions to easily manipulate those pixels and the RGBA values that they hold. An image in PIXL is represented as a matrix of pixels, which allows users to intuitively manipulate images using built in matrix operators and library functions for frequently used manipulations. Image processing has interesting applications to fields such as artificial intelligence and security, and therefore a language that makes image processing intuitive and accessible may be useful to those seeking to easily manipulate images.

In order to handle large image files and to produce a large number of images rapidly, PIXL has built-in garbage collection to ensure that memory is used efficiently and effectively. Built-in garbage collection allows the user to worry exclusively about image manipulation, with no need to concern themselves about memory management when working with large images. File IO allows PIXL users to easily import an image file, and just as easily export it after manipulation.

Through the use of an extensive standard library of functions, PIXL supports common image processing operations such as enhance_red, enhance_green, enhance_blue, gray scale, matrix AND, horizontal/vertical flip, and image cropping among others. This means that applying these operations is as easy as reading in a file, calling the standard library, and outputting the resulting image.

1.1 Language Goals

1.1.1 Intuitive Primitive Types and Operators

The main benefit of PIXL is that users can work with pixels and pixel matrices as primitive types. Pixels operators can add, subtract, and manipulate pixels in a number of ways. Images can be manipulated just as easily with simply matrix operators that apply pixel operations over the entirety of a pixel matrix. In just a few lines of code, an image's R, G, B, or A value can be increased by any specified amount. Filters like grayscale and negation are very easy to implement in PIXL.

1.1.2 Readability

PIXL's C like syntax means that type declaration and assignment are extremely straightforward and understandable. Anyone familiar with C-like languages will be able to understand PIXL. Additionally, since manipulating images is as simple as manipulating matrices, users who are familiar with matrices will find it easy to begin manipulating and processing images.

1.1.3 Memory Management

Because PIXL can be used with large image files and has the ability to output many images in a single program, PIXL uses built-in garbage collection to deal with memory allocation and deallocation so that the user doesn't need to. Therefore, the user can spend their efforts manipulating images rather than handling segmentation faults and memory leaks.

Chapter 2

Language Tutorial

2.1 Environment Setup

The PIXL compiler requires OCaml, LLVM, and a C compiler.

2.2 Using the Compiler

First, run the command `eval `opam config env`` in the project root.
Then, you can `make all`.
Optionally, run the `./testall.sh` script to validate your version of PIXL.

The included scripts to use PIXL work as follows for `your_program.p`:

```
./compile.sh your_program
./compileandrun.sh your_program
```

The compile script generates the binary `your_program`, while the compile and run script generates the binary, runs it, and then immediately deletes it. Note that you MUST LEAVE OUT THE `.p` EXTENSION when using these scripts to compile.

More advanced users that can read and understand these scripts can use their internals to create more complex systems in PIXL.

2.3 Building a Basic Program in PIXL

2.3.1 File Extension

All pixel files use the file extension ".p".

2.3.2 Parts of a PIXL Program

A PIXL Program is made up of the following components:

Global Variable Declarations
Function Declarations
    Local Variable Declarations
    Statements
        Expressions

2.3.3 Hello World

In this subsection, we will go over a simple program that prints "Hello World!". The following is the source code for helloworld.cm:

```
int main() {
    prints("Hello World");
}
```

int refers to the return type of the function. “main” is the name of the function. The function body contains a single statement that uses the function “prints” (print string) to print the string “Hello

World". The C like syntax and semantics of the language make PIXL accessible to a user familiar with C.

2.3.4 Variables

Variables are declared with a type name and variable name. Variable declarations occur at the top of a function, and declaration and assignment happen in two separate lines. The general form of variable declaration is shown below:

```
type varName;
varName = varValue;
```

An example of pixel declaration and assignment is shown below:

```
pixel p;
p = (255, 255, 255, 255);
```

The code snippet above declares and assigns a pixel. This particular pixel is white and fully opaque.

2.3.5 Functions

Functions in PIXL have the following general form:

```
returnType funcName(argType argName, argType argName, ...) {
/* function body */
}
```

2.3.6 Style and Organization

camelCase is the standard for variable and function declaration. Lines of code should not exceed 80 characters. Whitespace should be used to organize code and increase readability, and encapsulation should be used to organize and hide code.

2.3.7 Utilizing PIXL's Built-in Types

PIXL's most important built-in types are the pixel and the matrix. This section describes how to properly and effectively utilize these types.

Pixels

Pixels exist as a pointer to four consecutive integers in memory. Each of these integers is normalized to the range 0-255 if the user declares them outside this range. Anything above 255 gets floored to 255, and anything below 0 gets raised to 0. Any expression that evaluates to an integer can be used to initialize a pixel. The code below shows how to declare and instantiate a pixel in general.

```
pixel pixelName;
pixelName = (exprR, exprG, exprB, exprA);
```

Matrices

Matrices in Pixl are similar to 2-dimensional arrays in Java. They must contain lists of equal length length where every element is of the same type. They are declared and instantiated as follows:

```
int matrix m;
m = [1, 2, 3;4, 5, 6];
```

This creates an int matrix that looks like this:

```
| 1 2 3 |
| 4 5 6 |
```

Matrices can either be **int** matrices or **pixel** matrices.

2.3.8 Example Program

The program below – **flipPop.p** – takes an image, applies a set of transformations (enhance colors, invert - different depending on the pixel's position in the image)

```
int main() {
   pixel matrix inputMatrix;
   pixel matrix outputMatrix;
   int i;
   int j;
   pixel p;
   pixel matrix pmGray;
   int amt;

   inputMatrix = read("input.jpg");
   outputMatrix = |inputMatrix;

   for (i = 0; i < inputMatrix.rows; i=i+1)
   {
       for (j = 0; j < inputMatrix.cols; j=j+1)
       {
           p = inputMatrix[i][j];
           if (j>inputMatrix.cols/3) {
               if (i<inputMatrix.rows/3) {
                   outputMatrix[i][j] =
invert(enhanceRed(inputMatrix[i][j],i*70-j*20));
               }
               else if (j <= inputMatrix.cols*2/3) {
                   outputMatrix[i][j] =
enhanceGreen(inputMatrix[i][j], -i+j*3);
               }
               else {
                   outputMatrix[i][j] =
enhanceRed(inputMatrix[i][j], -i*2);
               }
           }
```

```
            else {
                if (i<inputMatrix.rows/3) {
                    outputMatrix[i][j] =
invert(enhanceRed(inputMatrix[i][j], 8*i+j*2));
                }
                else {
                    outputMatrix[i][j] =
enhanceRed(inputMatrix[i][j], 3*i*j);
                }
            }
        }
    }
    outputMatrix =
outputMatrix<<0:inputMatrix.rows/2,0:inputMatrix.cols-1>>;
    write(outputMatrix,"flipPop-out","jpg");
}
```

Chapter 3

Language Reference Manual

3.1 Tokens

PIXL has six classes of tokens: identifiers, keywords, constants, string literals, operators, and separators. Spaces, tabs and newlines can be used interchangeably to separate tokens; whitespace is required to separate what would otherwise be adjacent identifiers, keywords and constants.

3.1.2 Comments

Comments in PIXL are the same as those in MicroC. Characters /* introduce a comment and terminates with the characters */. Comments are ignored by the compiler.

3.1.3 Identifiers

Identifiers must have an uppercase or lowercase letter as a first character, which can be followed by any assortment of uppercase or lowercase letters, underscores, and numbers. Identifiers can be of any length and the letter case is significant.

3.1.4 Control Flow Keywords

The following are used as keywords and may not be used for any other purpose:

Keywords	**Description**
if	Enters statement if condition is met
else if	Paired with if; evaluated only when if/else if statements above in the same block haven't been satisfied
else	Default if no other conditions are satisfied in the block
for	Repeats until a condition is satisfied
while	Repeats until a condition is satisfied
return	Returns value from function
break	Exits loop containing statement and continues at the first statement outside the loop
main	The main function is the starting point of a program
true	Boolean literal for 1
false	Boolean literal for 0

void	Keyword used for functions without return values
matrix	Keyword used for declaring a new matrix
cols	Keyword used for accessing matrix column count
rows	Keyword used for accessing matrix row count
R, G, B, A	Keywords used for accessing pixel fields

3.2 Types

Each data type will be given a name and stored in a variable. These names are case-sensitive and made up of alphanumeric characters (Including '_'). Certain keywords used elsewhere in the language will be reserved and unable to be used as variable names ("int", "if", "for").

3.2.1 Basic Data Types

The basic data types listed will be implemented similarly to C. All basic types are immutable data types.

int

An Int is a 4 byte representation of an integer. An Int ranges in value from -2,147,483,648 to 2,147,483,647. When an Int overflows in either direction behavior is not defined.

Ints can be declared with or without an initial value. If no initial value is given it will have an undefined value until something is defined.

```
int x;
x = 5;
```

string

A string literal is denoted by a collection of characters bound by double quotes, as in "...". It is not possible to index through a statically declared string nor manipulate the data contained in the string.

void

Void is reserved as a type to allow functions to return nothing.

bool

A bool can have the value true or false

3.2.2 PIXL types

PIXL types represent algebraic structures that are useful for the types of programs PIXL is intended for. These types use

pixel

A pixel is the foundation for most pixel operations used in our language. Pixels are represented as a four-tuple of integers (or expressions that evaluate to integers) ranging inclusively from 0-255; separators used to delineate pixel values are indicated by commas, including optional whitespace.

The first three values correspond to a pixel's red, green, and blue value, respectively. The fourth tuple-value represents the pixel's alpha channel value, or the pixel's opacity.

```
pixel p;
p = (42, 42, 42, 42)
```

matrix

Matrices are fundamental to the functionality of the PIXL language. Pixel matrices are used to represent a 2D-array of values and can take expressions, pixel types and integers as arguments. A pixel matrix is defined inside brackets, with a semicolon ";" as a delimiter for rows and columns and a comma "," as a delimiter for the range of row/column values.

```
int pixel p;
pixel matrix pm1;
int matrix m2;

p = (42,42,42,42);
pm1 = [p, p; p, p];
m2 = [0,0;0,0];
```

3.3 Operators

3.3.1 Arithmetic/Logical Operators

Arithmetic/Logical Operator	Description
=	Assignment Operator
+	Additive Operator
-	Subtraction Operator
*	Multiplication Operator
/	Division Operator
==	Returns 1 if values are equal, 0 otherwise
!=	Returns 1 if values are not equal, 0 otherwise
>	Greater than operator
<	Less than operator
>=	Greater than or equal to operator
<=	Less than or equal to operator
&&	Logical AND operator
||	Logical OR operator
!	Logical NOT operator

3.3.2 Pixel Operators

Operator	**Description**	**Example**

+	+ works the same way as Java. However, when adding two pixels together you add the corresponding r,g,b,a values in each pixel together to create a new tuple. If the sum of two corresponding values exceeds 255, then 255 is used as the sum value. + can also be used as an operand between two matrices of the same dimensions: the + operator is applied to each corresponding pixel pair and adds them using the pixel + operator.	`pixel p1;`
		`pixel p2;`
		`pixel p3;`
		`p1 =`
		`(100,100,200,200);`
		`p2 =`
		`(50,50,100,100);`
		`p3 = p1 + p2;`
		
		`Result:`
		`p3:`
		`(150,150,255,255)`

-	Works the same way as addition, except you subtract the two tuples. Absolute value is used to avoid negative integers. - can also be used as an operation on matrices. Like addition, each corresponding pixel pair is subtracted.	`pixel p1;`
		`pixel p2;`
		`pixel p3;`
		
		`p1 =`
		`(100,100,200,200);`
		`p2 =`
		`(50,50,100,100);`
		`p3 = p1 - p2;`
		
		`Result:`
		`p3: (50,50,100,100);`

=	The equals assignment operator sets the value of the left variable equal to the value of the right side.	`pixel p1;`
		`pixel p2;`
		`p1 =`
		`(100,50,100,255);`
		`p2 = p1;'`
		
		`Result:`
		`p2:`
		`(100,50,100,255);`

3.3.3 Matrix Operators

Operator	**Description**	**Example**

~	~ stands for horizontal flip. This operator reverses the items contained in the rows of the matrix.	`int matrix m1;`
		`int matrix m2;`
		
		`m1 = [1,2;3,4];`
		`m2 = ~m1;`
		
		`Result:`
		`m2: [2,1;4,3]`

|	| stands for vertical flip. This operator reverses the items contained in the columns of the matrix.	`int matrix m1;`
		`int matrix m2;`
		
		`m1 = [1,2;3,4];`
		`m2 = |m1;`
		
		`Result:`
		`m2: [3,4;1,2]`

&&	The && operator works between pixel matrices. For each location (i,j), the AND compares the pixel value in the two matrices. If the two pixels are the same, the operator changes the pixel in the output matrix to (0,0,0,0). If the pixel values are different, the output matrix evaluates to the value of the left matrix in the operator.	`pixel matrix m1;`
		`pixel matrix m2;`
		`pixel matrix m3;`
		
		`m1 =`
		`[(100,200,300,40),(10`
		`0,200,150,50);(50,50,`
		`75,10),(60,60,60,60)]`
		
		`m2 =`
		`[(100,200,300,40),(15`
		`0,200,150,30);(50,50,`
		`75,10),(50,60,60,70)]`
		
		`m3 = m1 && m2;`
		
		`Result:`
		`m3:`
		`[(0,0,0,0),(100,200,1`
		`50,50);`
		`(0,0,0,0),(60,60,60,6`
		`0)]`

<< >>	The << >> operator stands for crop. This operator takes in four values v1,v2,v3,v4:
	
	<v1,v2; v3,v4>
	
	v1 and v2 stand for which rows to include (v1 inclusive and v2 exclusive), and v3 and v4 stand for which columns to include (v3 inclusive and v4 exclusive)
		`int matrix m1;`
		`int matrix m2;`
		
		`m1 = [1,2,3; 4,5,6; 7,8,9];`
		`m2 = m1<<0,2;1,3>>;`
		
		`Result:`
		`m3: [2,3;5,6]`

3.3.4 Operator Precedence

The following table displays operator precedence from highest to lowest.

Operator Symbol	Description
!	Logical NOT operator
* /	Multiply, divide
+ -	Add, subtract
> < >= <=	Greater than, less than, greater than or equal to, less than or equal to
== !=	Equality, inequality
&& ||	Logical AND, logical OR
=	Assignment operator

3.4 Functions and Variables

3.4.1 Function Declarations

Function headers are made in the same way as java. The return type is mentioned first, or void is used if there is no return type. Then, the function name is written, followed by 0 or more parameters.

A function may return any type—arithmetic, boolean, pixel, matrix.

```
type functionName(arguments){...}
```

3.4.2 Function Calls

A function can be called by including another file that contains the function, and then calling the function by using the following syntax:

3.4.3 Variable Declarations

```
type variable = new type();
```

3.4.4 Postfix/Prefix Expressions

```
i++; // increments i by 1, returns i
```

3.4 Scope

Identifiers can fall into one of two non-intersecting name-categories that do not interfere with each other: functions and variables. As such the same identifier may be used as a function name and a variable name. Functions may not have the same identifier.

Variables in different scopes may have the same identifier; however, variable identifiers in nested scopes can only refer to a single object. The scope of the identifiers is defined within the region of code in which they are declared, denoted by curly braces { … }. The lifetime of the identifier is therefore defined by its scope.

3.5 Statements

3.5.1 Blocks

Code blocks are 1 or more lines of code which are surrounded by curly braces. They are most commonly used in conditionals, loops and methods.

Note: Code blocks have local scope, so variables declared within a code block are only available within that code block.

3.5.2 Conditional statements

Conditional statements include **if**, **else if** and **else** statements, which work as they do in Java. Conditional statements must also contain a code block using braces to be executed if the condition is met.

```
if (condition) { // code to execute }
```

Example of conditional statement:

```
if (j > inputMatrix.cols/3) {
     if (i < inputMatrix.rows/3) {
          outputMatrix[i][j] = invert(inputMatrix[i][j]);
     }
     else if (j <= inputMatrix.cols*2/3){
          outputMatrix[i][j] = enhanceGreen(inputMatrix[i][j],-100);
     }
```

3.6 Loops

3.6.1 For Loops

For loops can consist of two types.
Standard for-loop: This is a standard for loop. It requires an initialization statement, a conditional statement and a code block to execute.
Example:

```
for (i = 0; i < inputMatrix.rows; i=i+1)
{
      for (j = 0; j < inputMatrix.cols; j=j+1)
      {
            p = inputMatrix[i][j];
            average = (p.R + p.G + p.B)/3;
      }
}
```

3.6.2 While Loops

This kind of loop evaluates a condition, and if it is true, executes the code block following the condition. At the end of the code block, it returns to the line with the condition, and repeats the process until the condition is no longer true.

Example:

```
while (foo < bar) {
      // code to execute
}
```

3.7 Return Statement

Return statements are used in methods to return a value. A return statement ends the method. The lines following a return statement are not executed.

All methods with a return type other than void must have a return statement. The return type must match the method signature.

Example:

```
int getRedValue() {
      // some code to execute
      return red;
}
```

Chapter 4

Project Plan

4.1 Planning and Scheduling

Once we decided that we were building a pixel processing language, our team met one to two times a week for the entire semester. We also met with our TA, Heather, who kept us on track and made sure we were hitting all of our milestones.

4.2 Development Process

Initially, the biggest challenge was figuring out how our grammar was going to fit into the parser, ast, and scanner files. Using the corresponding files written for microc, we were able to figure out how the pieces of the frontend fit together and could finally move on to cracking the semant and codegen files.

Figuring out how everything worked in the semant and codegen files required actually getting our hands dirty by beginning to implement features. Our first project feature after “Hello World” was the pixel primitive type. After figuring out the program flow from the frontend through semant and codegen, we each took one portion of the architecture (frontend, semant, codegen, testing, stdlib) and focused on it for the rest of the project.

Because of the way we broke up the work for this project, we needed to stay in touch throughout the week. To help us out in this regard, we used Github and messaging services, which helped us remain on the same page.

4.3 Testing

Testing was carried out continuously throughout the project: whenever new features were added, corresponding tests were added to the testing suite. Tests mainly focused on making sure semantic errors were being caught correctly in the semant file, as well as ensuring that newly implemented features worked properly.

4.4 Team Responsibilities

The team roles were allocated as follows:

Justin Borczuk - Codegen, LLVM Implementation
Jacob Gold - System Architecture, SAST Implementation, Semant Refactoring, C Function Linking, Environment Design
Maxwell Hu - Manager, Front-end Design
Shiv Sakhuja - Semant, Testing
Marco Starger - Testing, Standard Library Implementation, Language Guru

4.5 Project Timeline

Date	Milestones
October Week 3	Meetings and understanding MicroC
October Week 4	Meetings and allocating general roles
November - Week 1	Implemented basic types
November - Week 2	Hello world complete
November - Week 3	Implemented pixels as primitive type
November - Week 4	Implemented matrices as primitive type
December - Week 1	Implemented matrices as primitive type
December - Week 2	Implemented pixel and matrix operators
December - Week 3	SAST, operators, and garbage collection

4.6 Style Guide

PIXL style quite closely mimics that of C. All rules apply, with several new specific rules highlighted below:

Pixel Literals should be declared with no spaces between values and commas.

```
pixel p;
p = (0,0,0,0);    /* no spaces */
```

Matrix Literals should be declared with no spaces between values and commas.

```
int matrix m;
m = [0,0;0,0];    /* no spaces */
```

Unary matrix operators should be placed right next to the matrix ID. For example:

```
int matrix m1;
int matrix m2;
m1 = [0,0;0,0];
m2 = |m1;   /* vertical flip operator is placed on m1 without
spacing in between */
```

4.7 Project Log

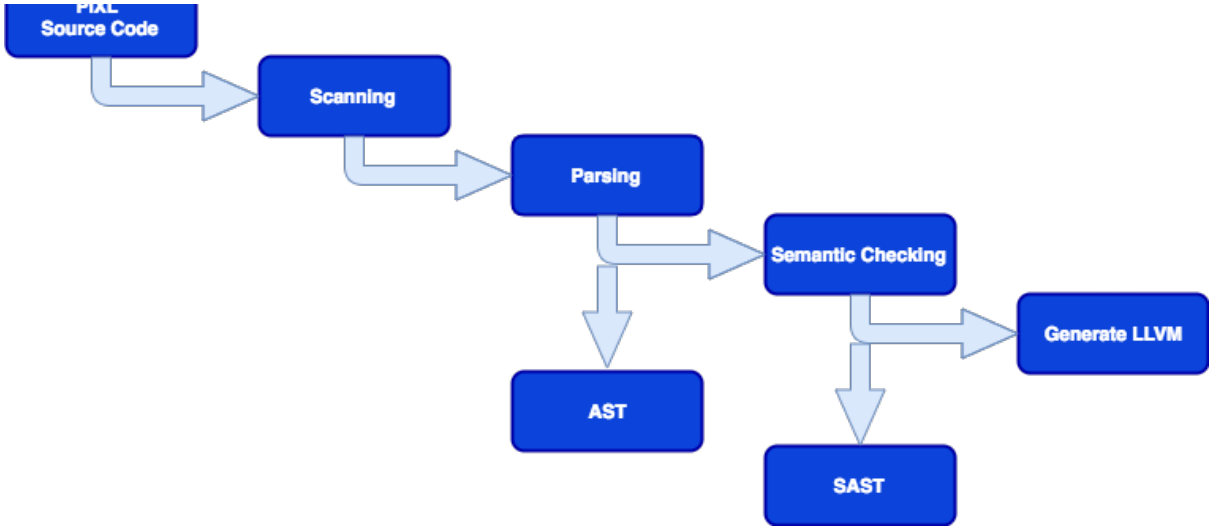
Below is our commit history on Github:

Chapter 5

System Architecture and Design

5.1 The Compiler

The Pixel compiler is built from the components diagrammed and detailed below, all implemented in OCaml. The main source file, pixl.ml, traverses the diagram, calling each of the components on the prior's output. After the LLVM code is generated, it is asserted to be valid by LLVM's own checker. Pixl.ml also allows for intermediate output of the AST and unchecked LLVM based on command line options. Before processing, the standard library (stdlib.p) is appended to the beginning of the input file.



5.1.1 Scanner (Max, Marco, Jacob, Justin)

The PIXL scanner, written in OCamllex, takes in the source file as a single large string of characters and turns them into tokens, using white space as a separator, based on the keywords and grammar of PIXL. Its output is a series of tokens.

5.1.2 Parser (Max, Jacob, Marco, Justin)

The PIXL parser, written in OCamlyacc takes in a series of tokens and creates an abstract syntax tree. The program is divided into a series of global variable declarations and a series of functions. Within the functions, the parser creates a list of statements, which can be control flow, blocks, or expressions that need to be evaluated. Its output is an Ast OCaml object, which is defined in a separate OCaml file.

5.1.3 Semantic Checker (Shiv, Jacob)

The PIXL semantic checker, written in OCaml, recursively traverses the AST and ensures that all statements and expressions have a valid semantic interpretation, and if successful converts it into an SAST, which is essentially an AST augmented with type data. Construction of the SAST also allowed us to reduce certain statements and expressions to others without having to write detailed LLVM implementations for each one. This process is most notably used for some of the more complex pixel and matrix operators, which are reduced to standard library calls in the semantic checker. The output of the semantic checker is an Sast OCaml object, which is defined in a separate OCaml file and semantically (in more than one sense) extends the Ast.

5.1.4 Code Generation (Justin, Marco, Jacob)

The PIXL code generation algorithm, implemented in OCaml, recursively traverses the SAST, and for each function builds LLVM instruction blocks using the LLVM OCaml library. In general, this traversal is post-order, building up from the construction of simple expressions to complex control flow statements.

5.2 C Libraries

Compiled PIXL programs are linked with stdlib.c, a C file that implements some key functions that are called by codegen to handle more complex procedures.

5.2.1 File I/O

PIXL's File I/O functions, read and write, are not actually implemented in PIXL. Calls to these functions are handled as a special case in code generation, which makes calls to functions written in stdlib.c that in turn use a public domain C image file I/O library called STB whose API meshes well with how pixel matrices are handled in PIXL. (The relevant library files are included as part of PIXL, but are also available on github.)

5.2.2 Basic String Functions

Also implemented in stdlib.c are the str_of_int and str_con functions, the latter of which is used to replace the + operator for strings.

5.3 Garbage Collection

As a side effect of using C to implement these functions, there is an opportunity for memory leaks to arise from pointers that are leftover from C. LLVM allows simple shadow stack-based garbage collection to be implemented, and when added all side effects of using these C functions disappear.

5.4 PIXL Libraries

As stated above, the PIXL libraries are implemented in the compiler by appending the necessary library code to the input file.

Chapter 6

Testing

6.1 Testing Phases

6.1.1 Basic Testing

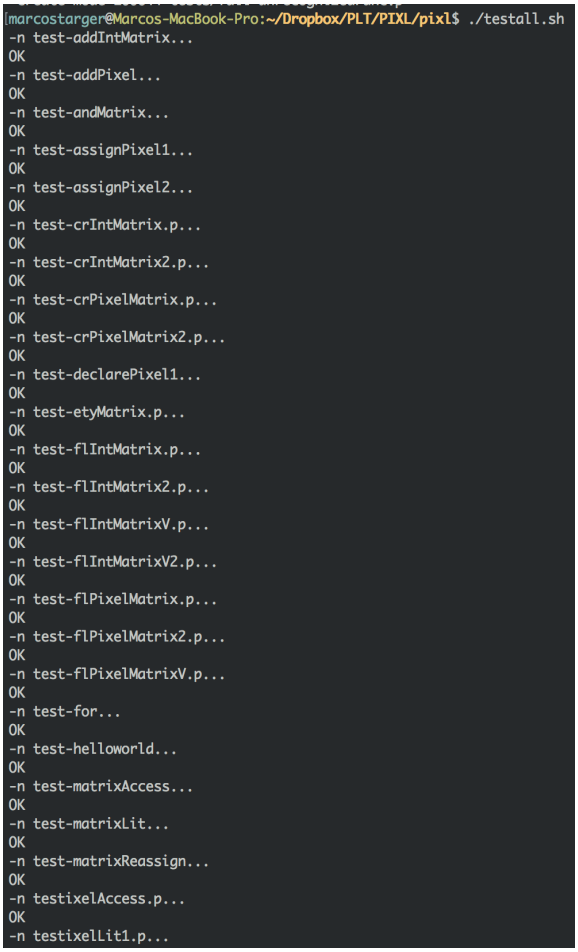
We started the testing suite with a few simple tests to test the basic functionality of our language, and to ensure that as we modified other components of our language, the basic functions of our language stayed intact. Here are two such files as an example:

```
test-returnX.p
int main() {
    int x;
    x = 1;
    print(x);
    return x;
}

test-printInt.p
int main() {
    int a;
    a = 5;
    print(a);
}
```

By running these basic tests we were always able to ensure that changes to our code didn't destroy the basic functionality of our language. We ensured that these basic functionality tests worked before all important commits and mergers.

Our testing suite passing all tests:



6.1.2 Integration Testing

Our test suite targeted the most important and critical functionality of our compiler. Every time a new feature was added, several tests were made to ensure that the feature worked properly, and that misuse of the feature would lead to the correct errors in semant.

We utilized the testing workflow from MicroC. Passing test files began with “test-”, so that the script knew to match the output with the corresponding .out file. Failing test files began with “fail-” so that the test script could match the error output to the corresponding .err file. The testing script allowed us to quickly and efficiently ensure that new features did not break old ones.

Every pull request was supposed to have passing tests for the entirety of the testing suite at that moment in time.

Tests were written and chosen to test every aspect of new features. As we built up more features, our testing suite grew and we would often find that old features broke. Our testing suite made it very easy to identify those breaks and to isolate exactly which features had been disrupted.

6.1.3 Testing Automation

We used a testing script called testall.sh that was modified from the original MicroC testall.sh script. This script ran all of our compiler integration tests. Testall could be run by simply typing

./testall.sh after running make, which compiled our compiler. As shown above, the script would then output OK if the test matched its corresponding .out or .err file.

6.1.4 Responsibilities

Marco was primarily responsible for the creation and maintenance of the testing suite, though all members contributed.

6.2 PIXL to LLVM IR

6.2.1 Example 1

```
test-printInt.p
int main() {
    int a;
    a=5;
    print(a);
}
```

LLVM:

```
; ModuleID = 'Pixl'
source_filename = "Pixl"

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%s\0A\00"

declare i32 @printf(i8*, ...)

declare i64* @read_img(i8*, ...)

declare i64 @write_img(i64*, i8*, i8*, ...)

declare i8* @str_of_int(i64, ...)

declare i8* @str_con(i8*, i8*, ...)

define i64 @main() gc "shadow-stack" {
entry:
  %a = alloca i64
  store i64 5, i64* %a
  %a1 = load i64, i64* %a
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i64 %a1)
  ret i64 0
}
```

6.2.2 Example 2

```
test-declarePixel.p
int main() {
      pixel p2;
      print(1);
}
```

LLVM:

```
; ModuleID = 'Pixl'
source_filename = "Pixl"

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%s\0A\00"

declare i32 @printf(i8*, ...)

declare i64* @read_img(i8*, ...)

declare i64 @write_img(i64*, i8*, i8*, ...)

declare i8* @str_of_int(i64, ...)

declare i8* @str_con(i8*, i8*, ...)

define i64 @main() gc "shadow-stack" {
entry:
  %p2 = alloca i64*
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i64 1)
  ret i64 0
}
```

6.2.3 Example 3

```
test-matrixAccess.p
int main() {
    int matrix m;
    m = [1,2;3,4];
    print(m[1][1]);
}
```

LLVM:

```
; ModuleID = 'Pixl'
source_filename = "Pixl"

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%s\0A\00"

declare i32 @printf(i8*, ...)

declare i64* @read_img(i8*, ...)

declare i64 @write_img(i64*, i8*, i8*, ...)

declare i8* @str_of_int(i64, ...)

declare i8* @str_con(i8*, i8*, ...)

define i64 @main() gc "shadow-stack" {
entry:
  %m = alloca i64*
  %0 = trunc i64 6 to i32
  %mallocsize = mul i32 %0, ptrtoint (i1** getelementptr (i1*,
i1** null, i32 1) to i32)
  %malloccall = tail call i8* @malloc(i32 %mallocsize)
  %matrix1 = bitcast i8* %malloccall to i64**
  %matrix2 = bitcast i64** %matrix1 to i64*
```

```
  store i64 2, i64* %pixel3
  %pixel31 = getelementptr i64, i64* %matrix2, i64 1
  store i64 2, i64* %pixel31
  %matrix3 = getelementptr i64, i64* %matrix2, i64 2
  store i64 1, i64* %matrix3
  %matrix32 = getelementptr i64, i64* %matrix2, i64 3
  store i64 2, i64* %matrix32
  %matrix33 = getelementptr i64, i64* %matrix2, i64 4
  store i64 3, i64* %matrix33
  %matrix34 = getelementptr i64, i64* %matrix2, i64 5
  store i64 4, i64* %matrix34
  store i64* %matrix2, i64** %m
  %m5 = load i64*, i64** %m
  %matrix7 = getelementptr i64, i64* %m5, i64 1
  %Access2 = load i64, i64* %matrix7
  %left = mul i64 %Access2, 1
  %add = add i64 %left, 3
  %matrix8 = getelementptr i64, i64* %m5, i64 %add
  %Access3 = load i64, i64* %matrix8
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i64 %Access3)
  ret i64 0
}

declare noalias i8* @malloc(i32)
```

Chapter 7

Lessons Learned

7.1 Justin

I learned a ton this semester working on a team. Firstly, I learned a lot about adjusting design choices on the spot. Because we laid out most of our design before actually starting to code the project, a lot of the features we laid out at the beginning were not possible to implement. That meant a lot of "audibles" in which we had to find work arounds and spontaneous solutions that were also effective.

Secondly, I became a better programmer through working on this project. Because I worked primarily in codegen with llvm implementation, I learned a lot about low level machine specific instructions. Pointers, which were never my favorite topic, became a must and if I didn't fully understand them I wouldn't have been able to implement our data types. Therefore, because there was such a huge learning curve to actually be able to implement changes to codegen, I learned a ton on the way.

Thirdly, and I think most importantly, I learned a lot about communication with regard to building projects. This was a massive project, and would not have gotten done without without a strong foundation centered in communication with teammates. If nothing else, I came out of this project a better communicator of ideas, issues, and solutions, a skill which I will value and utilize for the rest of my life.

7.2 Jacob

Working on PIXL this semester taught me a lot about both managing the synthesis of complex structures within a software system and about how the different areas of CS can align in completely unexpected ways.

Refactoring our semantic analysis to create the SAST was one of the most crucial tasks of the semester, and I pretty much had to write it from scratch after we realized that the intermediate representation would greatly speed up the rest of our workflow. But it was impossible to add functionality to either our front-end or code generation algorithms during the process due to conflicting APIs, so the project was extremely time-sensitive. Working with such an important component and refactoring it's outward representation taught me so much about our codebase as a whole and what really made me qualified to be our "Systems Architect." For me this really drilled down the whole "learn-by-doing" process and its importance in understanding software systems. It made me feel invincible, like I could just make a pull request for some complicated GitHub project and reimplement its entire internal logic flow.

About three quarters of the way through working on a project I had joined after the white paper was written, I was sitting on a toilet, thinking about what cool features we could implement, and realized that the image dataset augmentation algorithm I had read about for AI earlier that day could easily be implemented in our language given a few simply implemented operators. It was then that I realized no matter how much you feel the need to go deep into one area (in this case, language theory and compiler architecture), it is by becoming interdisciplinary that real progress is made.

7.3 Max

This project taught me an incredible amount about the collaborative and teamwork aspects of software development. It was the first curriculum-based large group programming project I had been a part of. One of the most challenging aspects (besides learning OCaml) of the project was the ability to work with four other team members on a schedule that was void of hard deadlines and learn how to coordinate those logistical issues.

From the technical side of things, I became much better at learning code not through APIs and modules or other sample code but from writing and testing code myself to learn OCaml. Staring at the unbelievably dense code did not help as much as initially writing and trying things out in MicroC. The project also made me appreciate how important testing was to the success of the project and also how difficult and puzzling it was to see tests failing for apparently no reason after pulling from our version control. Towards the end of the project, our comprehensive testing suite was crucial towards the success of the project and without it we would be of want for not only a finished product, but also a better understanding of our program.

From a group management perspective, one of the greatest challenges of working on such a large project with largely no external motivations for deadlines was to set discrete meeting times each week to work out problems and discuss ideas. Being disciplined about meeting with our TA and meeting with each other gave me a much greater appreciation for creating hard deadlines even when they weren't ascribed by other people. By prioritizing these meetings, we were able to get a better idea of how much we had to learn and at what rate we were learning. Of course as with most things (and especially in a project of this magnitude) we only hit our stride towards the last few weeks of the project and that was when it was most crucial to meet consistently despite increasingly large and urgent workloads. Overall, I learned a great deal from this class and from working on such a substantial group project; many of these lessons will carry on in mind towards the future.

7.4 Shiv

The semester-long project taught me a great deal about working on a large codebase collaboratively.

I learned about how compilers work under the hood (which really helps debug compiler errors!) and how to write a grammar for a programming language. After many mistakes and "28 shift/reduce errors", I learned about the importance of having a good unambiguous grammar and correct operator precedence and associativity. I also got used to more of the challenges that come with working on a large codebase with multiple contributors – merge conflicts, mysterious disappearance of chunks of code, etc. Because of the nature of our language, I also learned a great deal about pixel and image manipulations, especially while writing functions in the standard library. I learned about how the different color channels work, how alpha affects the pixel and how to combine and modify pixels in cool ways.

I think one of the things I realized from this project is that - although working in a group can sometimes be hard, it can also be incredibly productive. There were so many times where asking my teammates for help really sped up my work and, many instances where I should have asked for help, but didn't - which led to bottlenecks in my work. I think my biggest realization from this project was that: if the team is able to work well together, the shared knowledge base of everyone in the team is very good at pushing the project forward rapidly, i.e – "The whole is greater than the sum of its parts".

7.5 Marco

I've never worked on a group software project before, so there were many things that I was inexperienced in. First of all, though I had used Git in AP, I'd never needed to use it in a context where my commits and pushes would actually affect other team members, and therefore I lacked a lot of confidence in working with Git. Over time, the rest of my team helped me learn and gain confidence in using Git to handle version control. Second of all, I learned a lot about group dynamics for group projects. There were times when a bottleneck would prevent everyone from being able to work, and this came from overdependence on specialization. If we had 2-3 people to

than just one. I also realized that in a group of 5, not everyone was going to be able to meet at the same time or sometimes even in the same week. We came together best right before deadlines, and so perhaps we should have set our own internal deadlines in order to meet goals more quickly. Finally, I realized the importance of testing gradually and continuously throughout the project. Whenever we didn't keep up with adding tests, we weren't able to recognize when new features broke old ones. Overall, I had a great experience and learned a lot through this project.

Chapter 8

Code and Script Listing

pixl.ml

```
(* Author: Jacob Gold *)
(* Top-level of the Pixl compiler: scan & parse the input,
   check the AST and convert it into a SAST, generate LLVM
IR, and
   dump the .ll file *)

type action = Ast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);      (* Print the
AST only *)
                              ("-l", LLVM_IR);  (* Generate LLVM,
don't check *)
                              ("-c", Compile) ] (* Generate, check
LLVM IR *)
  else Compile in

  let file_to_string file =
    let array_string = ref [] in
      let ic = file in
          try
             while true do
                 array_string := List.append !array_string
[input_line ic]
              done;
              String.concat "\n" !array_string
          with End_of_file -> close_in ic; String.concat
"\n" !array_string

  in
  let in_file = open_in "stdlib.p" in
  let string_in = file_to_string in_file in
  let other_file = file_to_string stdin in
  let str = String.concat "\n" [other_file; string_in] in


  let lexbuf = Lexing.from_string str  in
  let ast = Parser.program Scanner.token lexbuf  in
  let sast = Semant.check ast in
  match action with
    Ast -> print_string (Ast.string_of_program ast)
   | LLVM_IR -> print_string (Llvm.string_of_llmodule
(Codegen.translate sast))
   | Compile -> let m = Codegen.translate sast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

scanner.mll

```
(* Authors: Maxwell Hu and Justin Borczuk *)
(* Ocamllex scanner for MicroC *)

{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRAC }
| ']'      { RBRAC }
| '|'      { BAR }
| '~'      { TILDA }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '.'      { DOT }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "<<"     { LANGLE }
| ">>"     { RANGLE }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "rows"   { ROWS }
| "cols"   { COLS }
| "R"      { RED }
| "G"      { GREEN }
| "B"      { BLUE }
| "A"      { ALPHA }
| "matrix" { MAT }
| "int"    { INT }
| "bool"   { BOOL }
| "void"   { VOID }
| "string" { STRING }
| "true"   { TRUE }
| "false"  { FALSE }
```

```
| '^'      { EXP }
| '"'      { read_string (Buffer.create 17) lexbuf }
| ":"      { COLON }
| "char"   { CHAR }
| "pixel"  { PIXEL }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm {
ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^
Char.escaped char)) }

(* sourced from
https://realworldocaml.org/v1/en/html/parsing-with-ocamllex-
and-menhir.html *)
and read_string buf =
  parse
  | '"'       { STR_LIT (Buffer.contents buf) }
  | '\\' '/'  { Buffer.add_char buf '/'; read_string buf
lexbuf }
  | '\\' '\\' { Buffer.add_char buf '\\'; read_string buf
lexbuf }
  | '\\' 'b'  { Buffer.add_char buf '\b'; read_string buf
lexbuf }
  | '\\' 'f'  { Buffer.add_char buf '\012'; read_string buf
lexbuf }
  | '\\' 'n'  { Buffer.add_char buf '\n'; read_string buf
lexbuf }
  | '\\' 'r'  { Buffer.add_char buf '\r'; read_string buf
lexbuf }
  | '\\' 't'  { Buffer.add_char buf '\t'; read_string buf
lexbuf }
  | [^ '"' '\\']+
    { Buffer.add_string buf (Lexing.lexeme lexbuf);
      read_string buf lexbuf
    }
  | _ { raise (Failure ("Illegal string character: " ^
Lexing.lexeme lexbuf)) }
  | eof { raise (Failure ("String is not terminated")) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

parser.mly

```
/* Authors: Maxwell Hu and Jacob Gold */
/* Ocamlyacc parser for Pixl */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT INCREMENT
DECREMENT
```

```
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE INT BOOL VOID STRING
%token LBRAC RBRAC COLON CHAR LANGLE RANGLE BAR TILDA
%token EXP PIXEL DOT ROWS COLS RED BLUE GREEN ALPHA MAT
%token <int> LITERAL
%token <string> ID
%token <string> STR_LIT
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc RED GREEN BLUE ALPHA
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT NEG TILDA BAR

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

decls:
   /* nothing */ { [], [] }
 | decls vdecl { ($2 :: fst $1), snd $1 }
 | decls fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list
stmt_list RBRACE
     { { typ = $1;
    fname = $2;
    formals = $4;
    locals = List.rev $7;
    body = List.rev $8 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    typ ID                   { [($1,$2)] }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ:
    INT                              { Int }
  | BOOL                             { Bool }
  | VOID                             { Void }
  | STRING                           { String }
```

```
  | PIXEL                           { Pixel }
  | typ MAT                         { Matrix($1) }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
    typ ID SEMI { ($1, $2) }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI
{ Expr $1 }
  | RETURN SEMI
{ Return Noexpr }
  | RETURN expr SEMI
{ Return $2 }
  | LBRACE stmt_list RBRACE
{ Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE
{ If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt
{ If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
{ For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt
{ While($3, $5) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    LITERAL                                                 {
Literal($1) }
  | TRUE                                                    {
BoolLit(true) }
  | FALSE                                                   {
BoolLit(false) }
  | ID                                                      {
Id($1) }
  | STR_LIT                                                 {
StringLit($1) }
  | expr PLUS   expr                                        {
Binop($1, Add,   $3) }
  | expr MINUS  expr                                        {
Binop($1, Sub,   $3) }
  | expr TIMES  expr                                        {
Binop($1, Mult,  $3) }
  | expr DIVIDE expr                                        {
Binop($1, Div,   $3) }
```

```
  | expr EQ     expr                                          {
Binop($1, Equal, $3) }
  | expr NEQ    expr                                          {
Binop($1, Neq,   $3) }
  | expr LT     expr                                          {
Binop($1, Less,  $3) }
  | expr LEQ    expr                                          {
Binop($1, Leq,   $3) }
  | expr GT     expr                                          {
Binop($1, Greater, $3) }
  | expr GEQ    expr                                          {
Binop($1, Geq,   $3) }
  | expr AND    expr                                          {
Binop($1, And,   $3) }
  | expr OR     expr                                          {
Binop($1, Or,    $3) }
  | MINUS expr %prec NEG                                      {
Unop(Neg, $2) }
  | NOT expr                                                  {
Unop(Not, $2) }
  | ID ASSIGN expr                                            {
Assign($1, $3) }
  | ID LBRAC expr RBRAC LBRAC expr RBRAC ASSIGN expr          {
Assignm($1, $3, $6, $9) }
  | ID DOT RED ASSIGN expr                                    {
Assignp($1, Red, $5) }
  | ID DOT GREEN ASSIGN expr                                  {
Assignp($1, Green, $5) }
  | ID DOT BLUE ASSIGN expr                                   {
Assignp($1, Blue, $5) }
  | ID DOT ALPHA ASSIGN expr                                  {
Assignp($1, Alpha , $5) }
  | ID LPAREN actuals_opt RPAREN                              {
Call($1, $3) }
  | LPAREN expr RPAREN                                        {
$2 }
  | LBRAC mat_lit RBRAC                                       {
MatrixLit(List.rev($2)) }
  | pixel_lit                                                 {
$1 }
  | ID DOT RED                                                {
Access($1, Red) }
  | ID DOT GREEN                                              {
Access($1, Green) }
  | ID DOT BLUE                                               {
Access($1, Blue) }
  | ID DOT ALPHA                                              {
Access($1, Alpha) }
  | ID LBRAC expr RBRAC LBRAC expr RBRAC                      {
MatrixAccess($1, $3, $6)}
  | ID LANGLE expr COLON expr COMMA expr COLON expr RANGLE    {
Crop($1, $3, $5, $7, $9) }
  | ID DOT ROWS                                               {
Rows($1) }
  | ID DOT COLS                                               {
Cols($1) }
```

```
  | BAR expr                                                    {
VFlip($2) }
  | TILDA expr                                                  {
HFlip($2) }
  | expr PLUS PLUS                                              {
Unop(Increment, $1) }
  | expr MINUS MINUS                                            {
Unop(Decrement, $1) }
  | MAT LPAREN expr COMMA expr COMMA typ RPAREN                  {
EMatrix($3, $5, $7) }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }

mat_lit:
    row_lit                       { [(List.rev $1)] }
  | mat_lit SEMI row_lit         { (List.rev $3) :: $1 }

row_lit:
    expr                            { [$1] }
  | row_lit COMMA expr             { $3 :: $1 }

pixel_lit:
    LPAREN expr COMMA expr COMMA expr COMMA expr RPAREN {
PixelLit($2, $4, $6, $8)
```

ast.ml

```
(* Authors: Maxwell Hu and Marco Starger*)
(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq
| Greater | Geq | And | Or
type field = Red | Blue | Green | Alpha

type uop = Neg | Not | Increment | Decrement
type flip = Bar | Underscore

type typ = Int | Bool | Void | String | Pixel | Char |
Matrix of typ

and expr =
    Literal of int
  | StringLit of string
  | BoolLit of bool
  | MatrixLit of expr list list
  | PixelLit of expr * expr * expr * expr
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
```

```
  | Assignp of string * field * expr
  | Assignm of string * expr * expr * expr
  | Call of string * expr list
  | Access of string * field
  | Crop of string * expr * expr * expr * expr
  | HFlip of expr
  | VFlip of expr
  | Noexpr
  | MatrixAccess of string * expr * expr
  | Rows of string
  | Cols of string
  | EMatrix of expr * expr * typ

type bind = typ * string

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
  }

type program = bind list * func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"
  | Increment -> "++"
  | Decrement -> "--"

let string_of_field = function
```

```
     Red -> "R"
   |  Blue -> "B"
   | Green -> "G"
   | Alpha -> "A"

let rec string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Void -> "void"
  | String -> "string"
  | Pixel -> "pixel"
  | Char -> "char"
  | Matrix(typ) -> string_of_typ typ ^ " matrix"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | MatrixLit(ll) -> "[" ^ String.concat "," (List.map
string_of_expr (List.concat ll)) ^ "]"
  | PixelLit(v1,v2,v3,v4) -> "(" ^ string_of_expr v1 ^ "," ^
string_of_expr v2 ^ "," ^ string_of_expr v3 ^ "," ^
string_of_expr v4 ^ ")"
  | StringLit(s) -> s
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^
string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr
el) ^ ")"
  | Access(id, field) -> id ^ "." ^ string_of_field field
  | Crop(v, e1, e2, e3, e4) -> v ^ "<" ^ string_of_expr e1 ^
":" ^ string_of_expr e2 ^ ", " ^ string_of_expr e3 ^ ":" ^
string_of_expr e4 ^ ">"
  | Noexpr -> ""
  | MatrixAccess(v, e1, e2) -> v ^ "[" ^ string_of_expr e1 ^
"]" ^ "[" ^ string_of_expr e2 ^ "]"
  | Rows(id) -> id ^ ".rows"
  | Cols(id) -> id ^ ".cols"
  | Assignp(id, f, e1) -> id ^ "." ^ string_of_field f ^ "="
^ string_of_expr e1
  | Assignm(id, e1, e2, value) -> id ^ "[" ^ string_of_expr
e1 ^ "]" ^ "[" ^ string_of_expr e2 ^ "]" ^ "=" ^
string_of_expr value
  | EMatrix(e1,e2,tp) -> "matrix(" ^ string_of_expr e1 ^ ",
" ^ string_of_expr e2 ^ "," ^ string_of_typ tp ^ ")"
  | HFlip(e) -> "hflip " ^ string_of_expr e
  | VFlip(e) -> "vflip " ^ string_of_expr e


let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt
```

```
stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n"
^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr
e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
string_of_stmt s


let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^
";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd
fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

semant.ml

```
(* Authors: Shiv Sakhuja and Jacob Gold *)
(* Semantic checking for the PIXL compiler *)

open Ast
open Sast
module E = Exceptions

module StringMap = Map.Make(String)

let report_duplicate exceptf list =
  let rec helper = function
      n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf
n1))
    | _ :: t -> helper t
    | [] -> ()
  in helper (List.sort compare list)


let rec check_binds exceptf = function
    (Void, s) :: _ -> raise (Failure(exceptf s))
  | _ :: tl        -> check_binds exceptf tl
  | _               -> ()
```

```
let check_function globals fdecls func =

  let _ = report_duplicate (fun n -> "duplicate formal " ^ n
^ " in " ^
    func.fname) (List.map snd func.formals) in
  let _ = check_binds (fun n -> "illegal void formal " ^ n ^
" in " ^
    func.fname) func.formals in
  let _ = report_duplicate (fun n -> "duplicate local " ^ n
^ " in " ^
    func.fname) (List.map snd func.locals) in
  let _ = check_binds (fun n -> "illegal void local " ^ n ^
" in " ^
    func.fname) func.locals in

  let symbols = List.fold_left (fun m (t, n) ->
StringMap.add n t m)
    StringMap.empty (globals @ func.formals @ func.locals )
  in

  let type_of_identifier s =
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("undeclared identifier
" ^ s))
  in

  let function_decl s = try StringMap.find s fdecls
       with Not_found -> raise (Failure ("unrecognized
function " ^ s))
  in

  let rec expr_to_sexpr e = (match e with
      Literal x               -> SLiteral(x, Int)
    | BoolLit b               -> SBoolLit(b, Bool)
    | PixelLit(x1,x2,x3,x4)   -> SPixelLit(expr_to_sexpr
x1,expr_to_sexpr x2,expr_to_sexpr x3,expr_to_sexpr x4,Pixel)
    | MatrixLit m             -> (check_matrix m)
    | Id s                    -> SId(s, type_of_identifier
s)
    | StringLit s             -> SStringLit(s, String)
    | Access(v,f)             -> (check_access v f) (*TODO*)
    | Binop(e1, op, e2)       -> (check_binop e1 op e2)
    | Unop(op, e)             -> (check_unop op e)
    | Noexpr                  -> SNoexpr
    | Assign(var, e)          -> (check_assign var e)
    | Assignp(var,field,e)    -> (check_assign_pixel var
field e)
    | Assignm(var,e1,e2,e3)    -> (check_assign_matrix var
e1 e2 e3)
    | Crop(var,r0,r1,c0,c1)   -> (check_crop var r0 r1 c0
c1)
    | Call(fname, actuals)    -> (check_call fname actuals)
    | MatrixAccess(var,e1,e2) -> (check_matrix_access var e1
e2)
    | Rows(var)               -> SRows(var)
```

```
    | EMatrix(e1,e2,e3)       -> (check_empty_matrix e1 e2
e3)
    | HFlip(e)                -> (check_h_flip e)
    | VFlip(e)                -> (check_v_flip e)
    | Cols(var)               -> SCols(var)
  )

  and check_h_flip e =
    let se = expr_to_sexpr e in
    let tp = sexpr_to_type se in
    (match tp with
      Matrix(Pixel) -> SCall("flipPixelMatrixH", [se],
Matrix(Pixel))
      | Matrix(Int) -> SCall("flipIntMatrixH", [se],
Matrix(Int))
      | _ -> raise(Failure("Can only flip int or pixel
matrices"))
    )

  and check_v_flip e =
    let se = expr_to_sexpr e in
    let tp = sexpr_to_type se in
    (match tp with
      Matrix(Pixel) -> SCall("flipPixelMatrixV", [se],
Matrix(Pixel))
      | Matrix(Int) -> SCall("flipIntMatrixV", [se],
Matrix(Int))
      | _ -> raise(E.IncorrectFlipExpr("Can only flip int or
pixel matrices"))
    )

  and check_empty_matrix e1 e2 tp =
  let se1 = expr_to_sexpr e1 in
  let se2 = expr_to_sexpr e2 in
  SEMatrix(se1, se2, Matrix(tp))

  and check_matrix_access var e1 e2 =
    let se1 = expr_to_sexpr e1 in
    let se2 = expr_to_sexpr e2 in
    let tp = type_of_identifier var in
    (match tp with
      Matrix(m_typ) -> SMatrixAccess(var,se1,se2,m_typ)
      | _ -> raise(E.InvalidMatrixAccess("Cannot access
elements of non-matrix type"))
    )

  and check_assign_pixel var field exp =
    let sexp = expr_to_sexpr exp in
    let tp = sexpr_to_type sexp in
    if type_of_identifier var != Pixel then
raise(Failure("Trying to assign to a pixel but found a " ^
string_of_typ (type_of_identifier var)))
    else if tp != Int then
raise(E.NonIntPixelReassignment("Pixel reassignment requires
an Int value"))
    else SAssignp(var,field,sexp,Int)
```

```
  and check_assign_matrix var e1 e2 e3 =
    let se1 = expr_to_sexpr e1 in
    let se2 = expr_to_sexpr e2 in
    let se3 = expr_to_sexpr e3 in
    let tp = sexpr_to_type se3 in
    SAssignm(var,se1,se2,se3,tp)

  and check_crop var r0 r1 c0 c1 =
    let svar = SId(var, type_of_identifier var) in
    let sr0 = expr_to_sexpr r0 in
    let sr1 = expr_to_sexpr r1 in
    let sc0 = expr_to_sexpr c0 in
    let sc1 = expr_to_sexpr c1 in
    if (r1 < r0) then raise (E.InvalidCropDimensions("Max
row must be greater than or equal to min row."))
    else if (c1 < c1) then raise
(E.InvalidCropDimensions("Max column must be greater than or
equal to min column."))
    else (match (type_of_identifier var) with
      Matrix(Pixel) -> SCall("cropPixelMatrix", [svar; sr0;
sr1; sc0; sc1], Matrix(Pixel))
      | Matrix(Int) -> SCall("cropIntMatrix", [svar; sr0;
sr1; sc0; sc1], Matrix(Int))
      | _ -> raise (Failure("Cannot crop a non-matrix"))
    )

  and check_call fname actuals =
    let rec helper = function
        ([], []) -> []
      | (_, []) | ([], _) ->
raise(E.IncorrectNumberOfArguments("Incorrect number of
arguments in call to " ^ fname))
      | ((ft, _) :: formals, e :: actuals) ->
        let se = expr_to_sexpr e in
        let t = sexpr_to_type se in
        if ft = t then se :: helper (formals, actuals) else
          raise (E.IllegalArgument ("illegal actual argument
found " ^
          string_of_typ t ^ " expected " ^ string_of_typ ft
^ " in " ^
          string_of_expr e))
    in
    let fd = function_decl fname in
    let formals = fd.formals in
    let sactuals = helper (formals, actuals) in
    SCall(fname, sactuals, fd.typ)

  and check_access var f =
    if (type_of_identifier var = Pixel) then
SAccess(var,f,Int)
    else raise(E.InvalidPixelAccess("Cannot call field
functions (R/G/B/A) on non-Pixel variables"))

  and check_binop e1 op e2 =
    let se1 = expr_to_sexpr e1 in
```

```
    let se2 = expr_to_sexpr e2 in
    let t1 = sexpr_to_type se1 in
    let t2 = sexpr_to_type se2 in
    (match op with
      Add | Sub | Mult | Div when t1 = Int && t2 = Int ->
SBinop(se1,op,se2,Int)
      | Equal | Neq when t1 = t2 -> SBinop(se1,op,se2,Bool)
      | Less | Leq | Greater | Geq when t1 = Int && t2 = Int
-> SBinop(se1,op,se2,Bool)
      | And | Or when t1 = Bool && t2 = Bool ->
SBinop(se1,op,se2,Bool)
      | Add when t1 = String && t2 = String ->
SCall("str_con", [se1; se2], String)
      | Add when t1 = Pixel && t2 = Pixel ->
SCall("addPixel", [se1;se2], Pixel)
      | Sub when t1 = Pixel && t2 = Pixel ->
SCall("subtractPixel", [se1;se2], Pixel)
      | Add when t1 = Matrix(Int) && t2 = Matrix(Int) ->
SCall("addIntMatrix", [se1;se2], Matrix(Int))
      | Sub when t1 = Matrix(Int) && t2 = Matrix(Int) ->
SCall("subtractIntMatrix", [se1;se2], Matrix(Int))
      | And when t1 = Matrix(Pixel) && t2 = Matrix(Pixel) ->
SCall("matrixAnd", [se1;se2], Matrix(Pixel))
      | _ -> raise (Failure ("illegal binary operator " ^
        string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
        string_of_typ t2 ^ " in " ^ string_of_expr e1 ^
string_of_op op ^ string_of_expr e2))
    )

  and check_matrix m =
    let add_if_match_1 l e =
      let se = expr_to_sexpr e in
      match l with
          []      -> (List.append l [se])
        | hd :: _ ->
         let t1 = sexpr_to_type hd in
         let t2 = sexpr_to_type se in
         if t1 = t2 then (List.append l [se]) else
raise(E.InconsistentMatrixTypes("MatrixLit types
inconsistent"))
      in
    let add_if_match_2 m l =
      let sl = List.fold_left add_if_match_1 [] l in
      match m with
          []       -> List.append m [sl]
        | [] :: _ ->
          if (List.length sl = 0)
          then List.append m [sl] else
raise(Failure("MatrixLit has lists of uneven length"))
        | (hd :: _) :: _ ->
          if (List.length (List.hd m)) != (List.length sl)
          then raise(E.UnevenMatrix("MatrixLit has lists of
uneven length"))
          else
          let t1 = sexpr_to_type hd in
          let t2 = sexpr_to_type (List.hd sl) in
```

```
        if t1 = t2 then List.append m [sl] else
raise(Failure("MatrixLit types inconsistent"))
      in
    let sm = List.fold_left add_if_match_2 [] m in
    let t = sexpr_to_type(List.hd(List.hd sm)) in
    if t != Int && t != Pixel then
raise(E.InvalidMatrixType("MatrixLit must be of type Int or
Pixel"))
    else SMatrixLit(sm, Matrix(t))

  and check_unop op e =
    let se = expr_to_sexpr e in
    let t = sexpr_to_type se in
    (match op with
        Neg when t = Int -> SUnop(op, se, Int)
      | Not when t = Bool -> SUnop(op, se, Bool)
      | Increment when t = Int -> SCall("increment",[se],
Int)
      | Decrement when t = Int -> SCall("decrement",[se],
Int)
      | _ -> raise (Failure ("illegal unary operator " ^
string_of_uop op ^
          string_of_typ t ^ " in " ^ string_of_expr e)))

  and check_assign var e =
    let lvaluet = type_of_identifier var in
    let se = expr_to_sexpr e in
    let rvaluet = sexpr_to_type se in
    let err = (E.IllegalAssignment("Illegal assignment
trying to assign " ^ string_of_expr e ^ " to " ^ var ^ "\n"
^ string_of_typ lvaluet ^ " = " ^
      string_of_typ rvaluet ^ " in " ^ string_of_expr e)) in
    let _ = (match lvaluet with
      Matrix(lt) -> (match rvaluet with
        Matrix(rt) -> if rt = lt then lvaluet else raise err
      | _ -> raise err
      )
    | _ -> if lvaluet = rvaluet then lvaluet else raise err
    ) in
    SAssign(var, se, lvaluet)

  and stmt_to_sstmt = (function
      Block sl              -> (check_block sl)
    | Expr e                -> (check_expr e)
    | Return e              -> (check_return e)
    | If(e, s1, s2)         -> (check_if e s1 s2)
    | While(e, s)           -> (check_while e s)
    | For(e1,e2,e3,s)       -> (check_for e1 e2 e3 s)
  )

  and check_block sl =
    let rec helper = (function
        (*Return _ :: a when a != [] -> raise (Failure
"nothing may follow a return")*)
      | hd :: tl                  -> (stmt_to_sstmt hd) ::
(helper tl)
```

```
      | []                          -> []
    )
    in
    let ssl = helper sl in
    SBlock(ssl)

  and check_expr e =
    let se = expr_to_sexpr e in
    let t = sexpr_to_type se in
    SExpr(se,t)

  and check_return e =
    let se = expr_to_sexpr e in
    let t = sexpr_to_type se in
    if t = func.typ then SReturn(se)
    else raise (Exceptions.InvalidReturnType ("return gives
" ^ string_of_typ t ^ " expected " ^
                          string_of_typ func.typ ^ " in " ^
string_of_expr e))

  and check_if e s1 s2 =
    let se = expr_to_sexpr e in
    let t = sexpr_to_type se in
    let ss1 = stmt_to_sstmt s1 in
    let ss2 = stmt_to_sstmt s2 in
    if t = Bool then SIf(se, ss1, ss2)
    else raise (E.InvalidConditional ("expected Boolean
expression in " ^ string_of_expr e))

  and check_while e s =
    let se = expr_to_sexpr e in
    let t = sexpr_to_type se in
    let ss = stmt_to_sstmt s in
    if t = Bool then SWhile(se, ss)
    else raise (E.InvalidConditional ("expected Boolean
expression in " ^ string_of_expr e))

  and check_for e1 e2 e3 s =
    let se1 = expr_to_sexpr e1 in
    let se2 = expr_to_sexpr e2 in
    let se3 = expr_to_sexpr e3 in
    let t = sexpr_to_type se2 in
    let ss = stmt_to_sstmt s in
    if t = Bool then SFor(se1, se2, se3, ss)
    else raise (E.InvalidConditional ("expected Boolean
expression in " ^ string_of_expr e2))

  and sexpr_to_type sexpr = match sexpr with
      SLiteral(_, typ)                -> typ
    | SStringLit(_, typ)              -> typ
    | SBoolLit(_, typ)                -> typ
    | SMatrixLit(_, typ)              -> typ
    | SPixelLit(_, _, _, _, typ)      -> typ
    | SBinop(_, _, _, typ)            -> typ
    | SUnop(_, _, typ)                -> typ
    | SId(_, typ)                     -> typ
```

```
    | SAssign(_, _, typ)                -> typ
    | SAssignp(_,_,_, typ)              -> typ
    | SAssignm(_,_,_,_, typ)            -> typ
    | SCall(_,_,typ)                    -> typ
    | SAccess(_,_,typ)                  -> typ
    | SMatrixAccess(_,_,_,typ)          -> typ
    | SRows(_)                          -> Int
    | SCols(_)                          -> Int
    | SNoexpr                           -> Void
    | SEMatrix(_,_,typ)                 -> typ

  in

  {
    styp = func.typ;
    sfname = func.fname;
    sformals = func.formals;
    slocals = func.locals;
    sbody = List.map stmt_to_sstmt func.body;
  }


let check (globals, functions) =
  let _ = report_duplicate (fun n -> "duplicate global " ^
n) (List.map snd globals) in
  let _ = check_binds (fun n -> "illegal void global" ^ n)
globals in
  let built_in_decls = StringMap.add "str_con"
    { typ = String; fname = "str_con"; formals = [(String,
"s1"); (String, "s2")];
      locals = []; body = [] } (StringMap.add "str_of_int"
    { typ = String; fname = "str_of_int"; formals = [(Int,
"i")];
      locals = []; body = [] } (StringMap.add "write"
    { typ = Int; fname = "write"; formals = [(Matrix(Pixel),
"m"); (String, "f"); (String, "e")];
      locals = []; body = [] } (StringMap.add "read"
    { typ = Matrix(Pixel); fname = "read"; formals =
[(String, "f")];
      locals = []; body = [] } (StringMap.add "print"
    { typ = Void; fname = "print"; formals = [(Int, "s")];
      locals = []; body = [] } (StringMap.singleton "prints"
     { typ = Void; fname = "print"; formals =
[(String,"s")];
       locals = []; body =  []  })))))
  in
  let check_functions m fdecl =
    if StringMap.mem fdecl.fname m then
      raise (Failure ("duplicate function " ^ fdecl.fname))
    else if StringMap.mem fdecl.fname built_in_decls then
      raise (Failure ("reserved function " ^ fdecl.fname))
    else StringMap.add fdecl.fname fdecl m
  in
  let fdecls = List.fold_left check_functions built_in_decls
functions in
  let   = try StringMap.find "main" fdecls
```

```
  with Not_found -> raise(Failure("no main function")) in
  let sfunctions = List.map (check_function globals fdecls)
functions in
  (globals, sfunctions)
```

sast.ml

```
(* Authors: Jacob Gold and Shiv Sakhuja *)
open Ast

type sexpr =
  SLiteral of int * typ
  | SStringLit of string * typ
  | SBoolLit of bool * typ
  | SMatrixLit of sexpr list list * typ
  | SPixelLit of sexpr * sexpr * sexpr * sexpr * typ
  | SId of string * typ
  | SBinop of sexpr * op * sexpr * typ
  | SUnop of uop * sexpr * typ
  | SAssign of string * sexpr * typ
  | SAssignp of string * field * sexpr * typ
  | SAssignm of string * sexpr * sexpr * sexpr * typ
  | SCall of string * sexpr list * typ
  | SAccess of string * field * typ
  | SMatrixAccess of string * sexpr * sexpr * typ
  | SNoexpr
  | SRows of string
  | SCols of string
  | SEMatrix of sexpr * sexpr * typ

type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr * typ
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt

type sfunc_decl = {
    styp : typ;
    sfname : string;
    sformals : bind list;
    slocals : bind list;
    sbody : sstmt list;
}
```

codegen.ml

```
(* Authors: Justin Borczuk and Marco Starger *)
(* Code generation: translate takes a SAST and produces LLVM
IR *)

module Semant = Semant
module L = Llvm
module A = Ast
module S = Sast
```

```
module StringMap = Map.Make(String)

let translate (globals, functions) =
  let context = L.global_context () in
  let the_module = L.create_module context "Pixl"
  and i64_t  = L.i64_type  context
  and i32_t  = L.i32_type  context
  and i8_t   = L.i8_type   context
  and i1_t   = L.i1_type   context
  and void_t = L.void_type context
  and str_t  = L.pointer_type (L.i8_type context) in

  let funcn = ref (List.hd functions) in

  let ltype_of_typ = function
      A.Int -> i64_t
    | A.Bool -> i1_t
    | A.Char -> i8_t
    | A.String -> str_t
    | A.Pixel -> L.pointer_type(L.i64_type context)
    | A.Matrix(_) -> L.pointer_type(L.i64_type context)
    | A.Void -> void_t in

  (* Declare each global variable; remember its value in a
map *)
  let global_vars =
    let global_var m (t, n) =
      let init = L.const_int (ltype_of_typ t) 0
      in StringMap.add n (L.define_global n init the_module)
m in
    List.fold_left global_var StringMap.empty globals in

  (* Declare printf(), which the print built-in function
will call *)
  let printf_t = L.var_arg_function_type i32_t [|
L.pointer_type i8_t |] in
  let printf_func = L.declare_function "printf" printf_t
the_module in

  let return_type = L.pointer_type(L.i64_type context) in
  let read_img = L.var_arg_function_type return_type [|
str_t |] in
  let read_img_func = L.declare_function "read_img" read_img
the_module in

  let arg1 = L.pointer_type(L.i64_type context) in
  let write_img = L.var_arg_function_type i64_t [|arg1;
str_t; str_t |] in
  let write_img_func = L.declare_function "write_img"
write_img the_module in

  let str_of_int = L.var_arg_function_type (L.pointer_type
i8_t) [| i64_t |] in
  let str_of_int_func = L.declare_function "str_of_int"
str of int the module in
```

```
  let str_con = L.var_arg_function_type (L.pointer_type
i8_t) [| L.pointer_type i8_t; L.pointer_type i8_t |] in
  let str_con_func = L.declare_function "str_con" str_con
the_module in

  (* Define each function (arguments and return type) so we
can call it *)
  let function_decls =
    let function_decl m fdecl =
      let name = fdecl.S.sfname
      and formal_types =
        Array.of_list (List.map (fun (t,_) -> ltype_of_typ
t) fdecl.S.sformals)
      in let ftype = L.function_type (ltype_of_typ
fdecl.S.styp) formal_types in
      StringMap.add name (L.define_function name ftype
the_module, fdecl) m in
    List.fold_left function_decl StringMap.empty functions
in

  (* Fill in the body of the given function *)
  let build_function_body fdecl =
    let (the_function, _) = StringMap.find fdecl.S.sfname
function_decls in
    let builder = L.builder_at_end context (L.entry_block
the_function) in
    let _ = L.set_gc (Some "shadow-stack") the_function in

    let int_format_str = L.build_global_stringptr "%d\n"
"fmt" builder in
    let str_format_str = L.build_global_stringptr "%s\n"
"fmt" builder in

    (* Construct the function's "locals": formal arguments
and locally
       declared variables.  Allocate each on the stack,
initialize their
       value, if appropriate, and remember their values in
the "locals" map *)
    let local_vars =
      let add_formal m (t, n) p = L.set_value_name n p;
        let local = L.build_alloca (ltype_of_typ t) n
builder in
        ignore (L.build_store p local builder);
        StringMap.add n local m in

      let add_local m (t, n) =
        let local_var = L.build_alloca (ltype_of_typ t) n
builder
        in StringMap.add n local_var m in

      let formals = List.fold_left2 add_formal
StringMap.empty fdecl.S.sformals
          (Array.to_list (L.params the_function)) in
      List.fold left add local formals fdecl.S.slocals in
```

```
    (* Return the value for a variable or formal argument *)
    let lookup n = try StringMap.find n local_vars
                   with Not_found -> StringMap.find n
global_vars
    in

    (* Invoke "f builder" if the current block doesn't
already
       have a terminal (e.g., a branch). *)
    let add_terminal builder f =
      match L.block_terminator (L.insertion_block builder)
with
        Some _ -> ()
      | None -> ignore (f builder) in

    (* Sast stmt builder and return the builder for each of
the statement's successor *)
    let rec expr builder = function
      S.SLiteral (i, _) -> L.const_int i64_t i
      | S.SBoolLit (b, _) -> L.const_int i1_t (if b then 1
else 0)
      | S.SStringLit (s, _) -> L.build_global_stringptr(s)
"strptr" builder
      | S.SNoexpr -> L.const_int i64_t 0
      | S.SEMatrix(rows,cols,typ) -> let rows = expr builder
rows
                                     and cols = expr builder
cols in
                                     (match typ with
                                     A.Matrix(A.Int) -> let
left = L.build_mul cols rows "left" builder in
                                                        let
mat = L.build_add left (L.const_int i64_t 2) "mat" builder
in
                                                        let
size = mat in
                                                        let
typ = L.pointer_type i64_t in
                                                        let
arr = L.build_array_malloc typ size "matrix1" builder in
                                                        let
arr = L.build_pointercast arr typ "matrix2" builder in
                                                        let
arr_ptr = L.build_gep arr [|L.const_int i64_t 0|] "pixel3"
builder in ignore(L.build_store (rows) arr_ptr builder);
                                                        let
arr_ptr = L.build_gep arr [|L.const_int i64_t 1|] "pixel3"
builder in ignore(L.build_store (cols) arr_ptr builder);
                                                        arr

                                     | A.Matrix (A.Pixel) ->
let left = L.build_mul cols rows "left" builder in
```

```
                                                                let
left = L.build_mul left (L.const_int i64_t 4) "left" builder
in
                                                                let
mat = L.build_add left (L.const_int i64_t 2) "mat" builder
in
                                                                let
size = mat in
                                                                let
typ = L.pointer_type i64_t in
                                                                let
arr = L.build_array_malloc typ size "matrix1" builder in
                                                                let
arr = L.build_pointercast arr typ "matrix2" builder in
                                                                let
arr_ptr = L.build_gep arr [|L.const_int i64_t 0|] "pixel3"
builder in ignore(L.build_store (rows) arr_ptr builder);
                                                                let
arr_ptr = L.build_gep arr [|L.const_int i64_t 1|] "pixel3"
builder in ignore(L.build_store (cols) arr_ptr builder);
                                                                arr
                                          | _ ->
raise(Failure("You shouldn't be seeing this")))

      | S.SAssignm(id, exp1, exp2, value, typ) -> let arr =
L.build_load (lookup id) id builder
                                                  and value =
expr builder value in
                                                  (match typ with
                                                  A.Pixel -> let
pointer = L.build_gep arr [|L.const_int i64_t 1|] "matrix7"
builder in
                                                                let
cols = L.build_load pointer "Access2" builder in
                                                                let
exp1 = expr builder exp1 in
                                                                let
exp2 = expr builder exp2 in
                                                                let
left = L.build_mul cols exp1 "left" builder in
                                                                let
left = L.build_add left exp2 "left2" builder in
                                                                let
left = L.build_mul left (L.const_int i64_t 4) "left3"
builder in
                                                                let
loc = L.build_add left (L.const_int i64_t 2) "right" builder
in
                                                                let
pointer1 = L.build_gep value [|L.const_int i64_t 0|]
"matrix8" builder in
                                                                let
num1 = L.build_load pointer1 "Access3" builder in
                                                                let
pointer2 = L.build_gep value [|L.const_int i64_t 1|]
"matrix8" builder in
```

```
                                                        let
num2 = L.build_load pointer2 "Access3" builder in
                                                        let
pointer3 = L.build_gep value [|L.const_int i64_t 2|]
"matrix8" builder in
                                                        let
num3 = L.build_load pointer3 "Access3" builder in
                                                        let
pointer4 = L.build_gep value [|L.const_int i64_t 3|]
"matrix8" builder in
                                                        let
num4 = L.build_load pointer4 "Access3" builder in
                                                        let
arr_ptr = L.build_gep arr [|loc|] "pixel3" builder in
ignore(L.build_store (num1) arr_ptr builder);
                                                        let
arr_ptr = L.build_gep arr [|L.build_add loc (L.const_int
i64_t 1) "add1" builder|] "pixel4" builder in
ignore(L.build_store (num2) arr_ptr builder);
                                                        let
arr_ptr = L.build_gep arr [|L.build_add loc (L.const_int
i64_t 2) "add2" builder|] "pixel5" builder in
ignore(L.build_store (num3) arr_ptr builder);
                                                        let
arr_ptr = L.build_gep arr [|L.build_add loc (L.const_int
i64_t 3) "add3" builder|] "pixel6" builder in
ignore(L.build_store (num4) arr_ptr builder);
                                                        arr


                                                | A.Int ->  let
pointer = L.build_gep arr [|L.const_int i64_t 1|] "matrix7"
builder in
                                                            let cols =
L.build_load pointer "Access2" builder in
                                                            let exp1 =
expr builder exp1 in
                                                            let exp2 =
expr builder exp2 in
                                                                 let
left = L.build_mul cols exp1 "left" builder in
                                                                 let
right = L.build_add exp2 (L.const_int i64_t 2) "right"
builder in
                                                                 let
loc = L.build_add left right "add" builder in
                                                                 let
pointer = L.build_gep arr [|loc|] "matrix8" builder in

ignore(L.build_store (value) pointer builder);
                                                                 arr
                                                | _ ->
raise(Failure("You shouldn't be seeing this")))
```

```
      | S.SAssignp(id,field,e1,_) -> let arr = L.build_load
(lookup id) id builder
                                          and value = expr
builder e1 in
                        (match field with
                                            A.Red ->   let
arr_ptr = L.build_gep arr [|L.const_int i64_t 0|] "pixel3"
builder in ignore(L.build_store (value) arr_ptr builder)

                                          | A.Green ->  let
arr_ptr = L.build_gep arr [|L.const_int i64_t 1|] "pixel3"
builder in ignore(L.build_store (value) arr_ptr builder)

                                          | A.Blue ->  let
arr_ptr = L.build_gep arr [|L.const_int i64_t 2|] "pixel3"
builder in ignore(L.build_store (value) arr_ptr builder)

                                          | A.Alpha ->  let
arr_ptr = L.build_gep arr [|L.const_int i64_t 3|] "pixel3"
builder in ignore(L.build_store (value) arr_ptr builder));

                        arr

      | S.SRows(id) ->    let arr = L.build_load (lookup id)
id builder in
                           let pointer = L.build_gep arr
[|L.const_int i64_t 0|] "pixel7" builder in
                           L.build_load pointer "Access1"
builder
      | S.SCols(id) ->    let arr = L.build_load (lookup id)
id builder in
                           let pointer = L.build_gep arr
[|L.const_int i64_t 1|] "pixel7" builder in
                           L.build_load pointer "Access1"
builder

      | S.SId (s, _) -> L.build_load (lookup s) s builder
      | S.SPixelLit(e1, e2, e3, e4, _) -> let size =
L.const_int i64_t 4 in
                               let typ = L.pointer_type i64_t
in
                               let arr = L.build_array_malloc
typ size "pixel1" builder in
                                             let arr =
L.build_pointercast arr typ "pixel2" builder in
                               let e1 = expr builder e1 in
                               let e2 = expr builder e2 in
                               let e3 = expr builder e3 in
                               let e4 = expr builder e4 in
                               let arr_ptr = L.build_gep arr
[|L.const_int i64_t 0|] "pixel3" builder in
ignore(L.build_store (e1) arr_ptr builder);
                               let arr_ptr = L.build_gep arr
[|L.const_int i64_t 1|] "pixel4" builder in
ignore(L.build_store (e2) arr_ptr builder);
```

```
                                let arr_ptr = L.build_gep arr
[|L.const_int i64_t 2|] "pixel5" builder in
ignore(L.build_store (e3) arr_ptr builder);
                                let arr_ptr = L.build_gep arr
[|L.const_int i64_t 3|] "pixel6" builder in
ignore(L.build_store (e4) arr_ptr builder);
                                arr

      | S.SMatrixLit(li, typ) -> (match typ with
                              A.Matrix(A.Pixel) -> let rows =
List.length li in
                              let columns = List.length
(List.hd li) in
                              let mat = (4 * rows * columns) +
2 in
                              let size = L.const_int i64_t mat
in
                              let typ = L.pointer_type i64_t in
                              let arr = L.build_array_malloc
typ size "matrix1" builder in
                              let arr = L.build_pointercast arr
typ "matrix2" builder in
                              let arr_ptr = L.build_gep arr
[|L.const_int i64_t 0|] "pixel3" builder in
ignore(L.build_store (L.const_int i64_t rows) arr_ptr
builder);
                              let arr_ptr = L.build_gep arr
[|L.const_int i64_t 1|] "pixel3" builder in
ignore(L.build_store (L.const_int i64_t (columns)) arr_ptr
builder);
                              for r=0 to rows-1 do
                                   for c=0 to columns-1 do
                                   let element = List.nth
(List.nth li r) c in
                                   let element = expr builder
element in
                                   let loc = (r * columns + c)
* 4 + 2 in
                                   let arr_ptr = L.build_gep
element [|L.const_int i64_t 0|] "matrix1" builder in
                                   let num1 = L.build_load
arr_ptr "num1" builder in
                                   let arr_ptr = L.build_gep
element [|L.const_int i64_t 1|] "matrix2" builder in
                                   let num2 = L.build_load
arr_ptr "num2" builder in
                                   let arr_ptr = L.build_gep
element [|L.const_int i64_t 2|] "matrix3" builder in
                                   let num3 = L.build_load
arr_ptr "num3" builder in
                                   let arr_ptr = L.build_gep
element [|L.const_int i64_t 3|] "matrix4" builder in
                                   let num4 = L.build_load
arr_ptr "num4" builder in
```

```
                                let arr_ptr = L.build_gep
arr [|L.const_int i64_t loc|] "matrix5" builder in
ignore(L.build_store (num1) arr_ptr builder);
                                let arr_ptr = L.build_gep
arr [|L.const_int i64_t (loc + 1)|] "matrix6" builder in
ignore(L.build_store (num2) arr_ptr builder);
                                let arr_ptr = L.build_gep
arr [|L.const_int i64_t (loc + 2)|] "matrix7" builder in
ignore(L.build_store (num3) arr_ptr builder);
                                let arr_ptr = L.build_gep
arr [|L.const_int i64_t (loc + 3)|] "matrix8" builder in
ignore(L.build_store (num4) arr_ptr builder);
                                done
                            done;
                            arr
                           | A.Matrix(A.Int) -> let rows =
List.length li in
                            let columns = List.length
(List.hd li) in
                            let mat = rows * columns + 2 in
                            let size = L.const_int i64_t mat
in
                            let typ = L.pointer_type i64_t in
                            let arr = L.build_array_malloc
typ size "matrix1" builder in
                            let arr = L.build_pointercast arr
typ "matrix2" builder in
                            let arr_ptr = L.build_gep arr
[|L.const_int i64_t 0|] "pixel3" builder in
ignore(L.build_store (L.const_int i64_t rows) arr_ptr
builder);
                            let arr_ptr = L.build_gep arr
[|L.const_int i64_t 1|] "pixel3" builder in
ignore(L.build_store (L.const_int i64_t columns) arr_ptr
builder);                                  for r=0 to rows-1 do
                                for c=0 to columns-1 do
                                let loc = r * columns + c +
2 in
                                let element = List.nth
(List.nth li r) c in
                                let element = expr builder
element in
                                let arr_ptr = L.build_gep
arr [|L.const_int i64_t loc|] "matrix3" builder in
                                ignore(L.build_store
(element) arr_ptr builder);
                                done
                            done;
                            arr
                           | _ -> raise(Failure("You shouldn't
be seeing this")))


      | S.SMatrixAccess(v,e1,e2,typ) -> (match typ with
                                    A.Pixel -> let arr1 =
L.build_load (lookup v) v builder in
```

```
                                    let pointer = L.build_gep
arr1 [|L.const_int i64_t 1|] "matrix7" builder in
                                    let cols = L.build_load
pointer "Access2" builder in
                                    let exp1 = expr builder
e1 in
                                    let exp2 = expr builder
e2 in
                                    let left = L.build_mul
cols exp1 "left" builder in
                                    let left = L.build_add
left exp2 "left2" builder in
                                    let left = L.build_mul
left (L.const_int i64_t 4) "left3" builder in
                                    let loc = L.build_add
left (L.const_int i64_t 2) "right" builder in
                                    let size = L.const_int
i64_t 4 in
                                    let typ = L.pointer_type
i64_t in
                                    let arr2 =
L.build_array_malloc typ size "pixel1" builder in
                                    let arr2 =
L.build_pointercast arr2 typ "pixel2" builder in
                                    let pointer1 =
L.build_gep arr1 [|loc|] "matrix8" builder in
                                    let num1 = L.build_load
pointer1 "Access3" builder in
                                    let pointer2 =
L.build_gep arr1 [|L.build_add loc (L.const_int i64_t 1)
"add1" builder|] "matrix8" builder in
                                    let num2 = L.build_load
pointer2 "Access3" builder in
                                    let pointer3 =
L.build_gep arr1 [|L.build_add loc (L.const_int i64_t 2)
"add2" builder|] "matrix8" builder in
                                    let num3 = L.build_load
pointer3 "Access3" builder in
                                    let pointer4 =
L.build_gep arr1 [|L.build_add loc (L.const_int i64_t 3)
"add3" builder|] "matrix8" builder in
                                    let num4 = L.build_load
pointer4 "Access3" builder in
                                    let arr_ptr = L.build_gep
arr2 [|L.const_int i64_t 0|] "pixel3" builder in
ignore(L.build_store (num1) arr_ptr builder);
                                    let arr_ptr = L.build_gep
arr2 [|L.const_int i64_t 1|] "pixel4" builder in
ignore(L.build_store (num2) arr_ptr builder);
                                    let arr_ptr = L.build_gep
arr2 [|L.const_int i64_t 2|] "pixel5" builder in
ignore(L.build_store (num3) arr_ptr builder);
                                    let arr_ptr = L.build_gep
arr2 [|L.const_int i64_t 3|] "pixel6" builder in
ignore(L.build_store (num4) arr_ptr builder);
                                    arr2
```

```
                                    | A.Int -> let arr =
L.build_load (lookup v) v builder in
                                    let pointer = L.build_gep
arr [|L.const_int i64_t 1|] "matrix7" builder in
                                    let cols = L.build_load
pointer "Access2" builder in
                                    let exp1 = expr builder
e1 in
                                    let exp2 = expr builder
e2 in
                                    let left = L.build_mul
cols exp1 "left" builder in
                                    let right = L.build_add
exp2 (L.const_int i64_t 2) "right" builder in
                                    let loc = L.build_add
left right "add" builder in
                                    let pointer = L.build_gep
arr [|loc|] "matrix8" builder in
                                    L.build_load pointer
"Access3" builder
                                    | _ -> raise(Failure("You
shouldn't be seeing this")))
      | S.SAccess(s, e, _) ->
          let arr = L.build_load (lookup s) s builder in
                            (match e with
                              A.Red  -> let pointer =
L.build_gep arr [|L.const_int i64_t 0|] "pixelRed" builder
in
                                        L.build_load pointer
"Access1" builder

                            | A.Green -> let pointer =
L.build_gep arr [|L.const_int i64_t 1|] "pixelGreen" builder
in
                                         L.build_load pointer
"Access1" builder

                            | A.Blue ->  let pointer =
L.build_gep arr [|L.const_int i64_t 2|] "pixelBlue" builder
in
                                         L.build_load pointer
"Access1" builder

                            | A.Alpha -> let pointer =
L.build_gep arr [|L.const_int i64_t 3|] "pixelAlpha" builder
in
                                         L.build_load pointer
"Access1" builder)


      | S.SBinop(e1, op, e2, _) ->
          let e1' = expr builder e1
          and e2' = expr builder e2 in
              (match op with
               A.Add     ->   L.build add
```

```
              | A.Sub     -> L.build_sub
              | A.Mult    -> L.build_mul
              | A.Div     -> L.build_sdiv
              | A.And     -> L.build_and
              | A.Or      -> L.build_or
              | A.Equal   -> L.build_icmp L.Icmp.Eq
              | A.Neq     -> L.build_icmp L.Icmp.Ne
              | A.Less    -> L.build_icmp L.Icmp.Slt
              | A.Leq     -> L.build_icmp L.Icmp.Sle
              | A.Greater -> L.build_icmp L.Icmp.Sgt
              | A.Geq     -> L.build_icmp L.Icmp.Sge
             ) e1' e2' "tmp" builder

      | S.SUnop(op, e, _) ->
          let e' = expr builder e in
          (match op with
            A.Neg       -> L.build_neg e' "tmp" builder
            | A.Not     -> L.build_not e' "tmp" builder
            | _ -> raise(Failure("You shouldn't be seeing
this")))

      | S.SAssign (s, e, _) -> let e' = expr builder e in
                           ignore (L.build_store e' (lookup
s) builder); e'

      | S.SCall ("print", [e], _) | S.SCall ("printb", [e],
_) ->
          L.build_call printf_func [| int_format_str ; (expr
builder e) |]
            "printf" builder
      | S.SCall ("prints", [e], _) ->
          L.build_call printf_func [| str_format_str ; (expr
builder e) |]
            "printf" builder
      | S.SCall ("read", [e], _) ->
          L.build_call read_img_func [| (expr builder e) |]
"read_img" builder
      | S.SCall ("write", [e1;e2;e3], _)->
          L.build_call write_img_func [|(expr builder e1);
(expr builder e2); (expr builder e3)|] "write_img" builder
      | S.SCall ("str_of_int", [e], _) ->
          L.build_call str_of_int_func [| (expr builder e)
|] "str_of_int" builder
      | S.SCall ("str_con", [e1;e2], _) ->
          L.build_call str_con_func [| (expr builder e1);
(expr builder e2) |] "str_con" builder
      | S.SCall(f, act, _) ->
          let (fdef, fdecl) = StringMap.find f
function_decls in
          let actuals = List.rev (List.map (expr builder)
(List.rev act)) in
          let result = (match fdecl.S.styp with
                                A.Void -> ""
                                | _ -> f ^ "_result") in
            L.build_call fdef (Array.of_list actuals) result
builder
```

```
      in

    let rec stmt builder = function
      S.SBlock sl -> List.fold_left stmt builder sl
      |S.SExpr (e, _) -> ignore (expr builder e); builder
      |S.SReturn (e) -> ignore (match !funcn.S.styp with
          A.Void -> L.build_ret_void builder
          | _ -> L.build_ret (expr builder e) builder);
builder
      |S.SIf (predicate, then_stmt, else_stmt) ->
         let bool_val = expr builder predicate in
         let merge_bb = L.append_block context "merge"
the_function in

         let then_bb = L.append_block context "then"
the_function in
         add_terminal (stmt (L.builder_at_end context
then_bb) then_stmt)
           (L.build_br merge_bb);

         let else_bb = L.append_block context "else"
the_function in
         add_terminal (stmt (L.builder_at_end context
else_bb) else_stmt)
           (L.build_br merge_bb);

         ignore (L.build_cond_br bool_val then_bb else_bb
builder);
         L.builder_at_end context merge_bb
      |S.SWhile (predicate, body) ->
          let pred_bb = L.append_block context "while"
the_function in
          ignore (L.build_br pred_bb builder);

          let body_bb = L.append_block context "while_body"
the_function in
          add_terminal (stmt (L.builder_at_end context
body_bb) body)
            (L.build_br pred_bb);

          let pred_builder = L.builder_at_end context
pred_bb in
          let bool_val = expr pred_builder predicate in

          let merge_bb = L.append_block context "merge"
the_function in
          ignore (L.build_cond_br bool_val body_bb merge_bb
pred_builder);
          L.builder_at_end context merge_bb
     | S.SFor (e1, e2, e3, body) -> stmt builder
           ( S.SBlock [S.SExpr(e1,A.Int); S.SWhile (e2,
S.SBlock [body ; S.SExpr(e3,A.Int)]) ] )

    in
    (* Build the code for each statement in the function *)
```

```
    let builder = stmt builder (S.SBlock fdecl.S.sbody) in

    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.S.styp with
        A.Void -> L.build_ret_void
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

  List.iter build_function_body functions;
  the_module
```

exceptions.ml

```
(* Author: Shiv Sakhuja *)
exception InvalidMatrixAccess of string
exception NonIntPixelReassignment of string
exception IncorrectFlipExpr of string
exception InvalidCropDimensions of string
exception IncorrectNumberOfArguments of string
exception IllegalArgument of string
exception InvalidPixelAccess of string
exception InconsistentMatrixTypes of string
exception UnevenMatrix of string
exception InvalidMatrixType of string
exception IllegalAssignment of string
exception InvalidReturnType of string
exception InvalidConditional of string
```

stdlib.p

```
(* Authors: Marco Starger and Shiv Sakhuja *)
int increment(int a) {
    return a+1;
}

int decrement(int a) {
    return a-1;
}

int max(int a, int b) {
    if (a < b) {
        return b;
    }
    else {
        return a;
    }
}

int abs(int x) {
    if (x > 0) {
        return x;
    }
    else {
        return -x;
    }
}
```

```
pixel copyPixel(pixel p) {
    int r;
    int g;
    int b;
    int a;
    pixel q;

    r = p.R;
    g = p.G;
    b = p.B;
    a = p.A;

    q = (r, g, b, a);
    return q;
}

pixel matrix cropPixelMatrix(pixel matrix pm, int r1, int
r2, int c1, int c2) {
    int i;
    int j;
    pixel p;
    pixel matrix pm2;

    pm2 = matrix(r2-r1, c2-c1, pixel);

    for (i = r1; i < r2; i=i+1)
    {
        for (j = c1; j < c2; j=j+1)
        {
            pm2[i-r1][j-c1] = copyPixel(pm[i][j]);
        }
    }

    return pm2;
}

int matrix cropIntMatrix(int matrix m, int r1, int r2, int
c1, int c2) {
    int i;
    int j;
    int a;
    int matrix m2;

    m2 = matrix(r2-r1, c2-c1, int);

    for (i = r1; i < r2; i=i+1)
    {
        for (j = c1; j < c2; j=j+1)
        {
            m2[i-r1][j-c1] = m[i][j];
        }
    }

    return m2;
}
```

```
int matrix flipIntMatrixH(int matrix m) {
    int height;
    int width;
    int i;
    int j;
    int matrix m2;
    height = m.rows;
    width = m.cols;

    m2 = matrix(m.rows, m.cols, int);

    for (i = 0; i < height; i=i+1)
    {
        for (j = 0; j < width; j=j+1)
        {
            m2[i][j] = m[i][width-1-j];
        }
    }

    return m2;
}

pixel matrix flipPixelMatrixH(pixel matrix pm) {
    int height;
    int width;
    int i;
    int j;
    pixel matrix pm2;
    height = pm.rows;
    width = pm.cols;

    pm2 = matrix(pm.rows, pm.cols, pixel);

    for (i = 0; i < height; i=i+1)
    {
        for (j = 0; j < width; j=j+1)
        {
            pm2[i][j] = copyPixel(pm[i][width-1-j]);
        }
    }

    return pm2;
}

int matrix flipIntMatrixV(int matrix m) {
    int height;
    int width;
    int i;
    int j;
    int matrix m2;
    height = m.rows;
    width = m.cols;

    m2 = matrix(m.rows, m.cols, int);
```

```
    for (i = 0; i < width; i=i+1)
    {
        for (j = 0; j < height; j=j+1)
        {
            m2[j][i] = m[height-1-j][i];
        }
    }

    return m2;
}

pixel matrix flipPixelMatrixV(pixel matrix pm) {
    int height;
    int width;
    int i;
    int j;
    pixel matrix pm2;
    height = pm.rows;
    width = pm.cols;

    pm2 = matrix(pm.rows, pm.cols, pixel);


    for (i = 0; i < width; i=i+1)
    {
        for (j = 0; j < height; j=j+1)
        {
            pm2[j][i] = copyPixel(pm[height-1-j][i]);
        }
    }

    return pm2;
}

bool pixelEquality(pixel p1, pixel p2) {
    if ((p1.R == p2.R) && (p1.G == p2.G) && (p1.B == p2.B)
&& (p1.A == p2.A))
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool pixelSimilarity(pixel p1, pixel p2) {
    int diff;
    diff = 30;
    if ((absInt(p1.R - p2.R) < diff) && (absInt(p1.G - p2.G)
< diff) && (absInt(p1.B - p2.B) < diff) && (absInt(p1.A -
p2.A) < diff))
    {
        return true;
    }
    else
```

```
    {
        return false;
    }
}

int absInt(int a)
{
    if (a < 0)
    {
        return -a;
    }
    else
    {
        return a;
    }
}

pixel matrix matrixAnd(pixel matrix pm1, pixel matrix pm2) {
    int i;
    int j;

    pixel matrix pm3;

    pm3 = matrix(pm1.rows, pm1.cols, pixel);

    for (i = 0; i < pm1.rows; i=i+1)
    {
        for (j = 0; j < pm1.cols; j=j+1)
        {
            if (pixelSimilarity(pm1[i][j], pm2[i][j]))
            {
                pm3[i][j] = (255,255,255,255);
            }

            else
            {
                pm3[i][j] = copyPixel(pm1[i][j]);
            }
        }
    }

    return pm3;
}

pixel matrix invertMatrix(pixel matrix pm) {
    int i;
    int j;

    pixel matrix pm1;
    pm1 = pm;

    for (i = 0; i < pm.rows; i=i+1)
    {
        for (j = 0; j < pm.cols; j=j+1) {
            pm1[i][j] = (255,255,255,255) - pm[i][j];
        }
```

```
    }

    return pm1;
}

pixel invert(pixel og) {
    pixel p;
    p = (255,255,255,255) - og;
    return p;
}

pixel addPixel(pixel p1, pixel p2) {
    pixel p3;
    p3 = (0,0,0,0);
    p3.R = p1.R + p2.R;
    p3.G = p1.G + p2.G;
    p3.B = p1.B + p2.B;
    p3.A = max(p1.A, p2.A);

    return p3;
}

pixel matrix grayscale(pixel matrix pm) {
    int i;
    int j;
    int average;
    pixel p;
    pixel matrix pmGray;

    pmGray = matrix(pm.rows, pm.cols, pixel);

    for (i = 0; i < pm.rows; i=i+1)
    {
        for (j = 0; j < pm.cols; j=j+1)
        {
            p = pm[i][j];
            average = (p.R + p.G + p.B)/3;
            pmGray[i][j] = (average, average, average, p.A);
        }
    }

    return pmGray;
}

pixel subtractPixel(pixel p1, pixel p2) {
    pixel p3;
    p3 = (0,0,0,0);
    p3.R = abs(p1.R - p2.R);
    p3.G = abs(p1.G - p2.G);
    p3.B = abs(p1.B - p2.B);
    p3.A = max(p1.A, p2.A);

    return p3;
}

int matrix addIntMatrix(int matrix a, int matrix b) {
```

```
    int matrix m;
    int i;
    int j;
    m = matrix(a.rows, a.cols, int);
    for (i = 0; i < a.rows; i=i+1)
    {
        for (j = 0; j < a.cols; j=j+1)
        {
            m[i][j] = a[i][j] + b[i][j];
        }
    }

    return m;
}

int matrix subtractIntMatrix(int matrix a, int matrix b) {
    int matrix m;
    int i;
    int j;
    m = matrix(a.rows, a.cols, int);
    for (i = 0; i < a.rows; i=i+1)
    {
        for (j = 0; j < a.cols; j=j+1)
        {
            m[i][j] = a[i][j] - b[i][j];
        }
    }

    return m;
}

pixel enhanceRed(pixel p, int amount) {
  int currentRed;
  int newRed;
  currentRed = p.R;
  newRed = currentRed + amount;
  if (newRed >= 255) {
      newRed = 255;
  }
  p.R = newRed;
  return p;
}

pixel enhanceGreen(pixel p, int amount) {
  int currentGreen;
  int newGreen;
  currentGreen = p.G;
  newGreen = currentGreen + amount;
  if (newGreen >= 255) {
      newGreen = 255;
  }
  p.G = newGreen;
  return p;
}

pixel enhanceBlue(pixel p, int amount) {
```

```
  int currentBlue;
  int newBlue;
  currentBlue = p.B;
  newBlue = currentBlue + amount;
  if (newBlue >= 255) {
      newBlue = 255;
  }
  p.B = newBlue;
  return p;
}

pixel matrix enhanceRedMatrix(pixel matrix m, int amount) {
    int i;
    int j;
    pixel p;
    int red;
    pixel matrix pm;
    pm = matrix(m.rows, m.cols, pixel);

    for (i = 0; i < m.rows; i=i+1)
    {
        for (j = 0; j < m.cols; j=j+1)
        {
            p = m[i][j];
            red = p.R;
            red = red + amount;
            p.R = red;
            pm[i][j] = p;
        }
    }

    return pm;

}

pixel matrix enhanceGreenMatrix(pixel matrix m, int amount)
{
    int i;
    int j;
    pixel p;
    int green;
    pixel matrix pm;
    pm = matrix(m.rows, m.cols, pixel);

    for (i = 0; i < m.rows; i=i+1)
    {
        for (j = 0; j < m.cols; j=j+1)
        {
            p = m[i][j];
            green = p.G;
            green = green + amount;
            p.G = green;
            pm[i][j] = p;
        }
    }
```

```
    return pm;

}


pixel matrix enhanceBlueMatrix(pixel matrix m, int amount) {
    int i;
    int j;
    pixel p;
    int blue;
    pixel matrix pm;
    pm = matrix(m.rows, m.cols, pixel);

    for (i = 0; i < m.rows; i=i+1)
    {
        for (j = 0; j < m.cols; j=j+1)
        {
            p = m[i][j];
            blue = p.B;
            blue = blue + amount;
            p.B = blue;
            pm[i][j] = p;
        }
    }

    return pm;

}
```

stdlib.c

```
#include <stdio.h>
#include <string.h>
#define STB_IMAGE_IMPLEMENTATION
#define STBI_ASSERT(x)
#include "stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

unsigned long* read_img(char* filename) {
    int w, h;
    unsigned char *data = stbi_load(filename, &w, &h, NULL,
4);
    size_t matrix_size = sizeof(long) * (2 + w * h * 4);
    unsigned long *img = malloc(matrix_size); //rip ram
    *img = (unsigned long)h;
    *(img + 1) = (unsigned long)w;
    unsigned long *data_start = img + 2;
    for (int i = 0; i < h * w * 4; i++) {
        *(data_start + i) = (unsigned long)(*(data + i));
    }
    stbi_image_free(data);
    return img;
}
```

```
long write_img(unsigned long* img, char* filename, char*
extension) {
    int h = (int)(*img);
    int w = (int)(*(img + 1));
    unsigned long *data_start = img + 2;
    unsigned char *data = malloc(sizeof(char) * 4 * w * h);
//gonna free this time!
    for (int i = 0; i < h * w * 4; i++) {
        unsigned long val = *(data_start + i);
        if ((long)val < 0l)
            val = 0ul;
        if (val > 255ul)
            val = 255ul;
        *(data + i) = (unsigned char)val;
    }
    int out;
    char* name = malloc(strlen(filename) + 5);
    strcpy(name, filename);
    strcat(name, ".");
    strcat(name, extension);
    if (!strcmp(extension, "png")) {
        out = stbi_write_png(name, w, h, 4, data, 0);
    } else if (!strcmp(extension, "bmp")) {
        out = stbi_write_bmp(name, w, h, 4, data);
    } else if (!strcmp(extension, "tga")) {
        out = stbi_write_tga(name, w, h, 4, data);
    } else if (!strcmp(extension, "jpg")) {
        out = stbi_write_jpg(name, w, h, 4, data, 100);
    } else {
        return 1;
    }
    free(name);
    free(data);
    if (out) {
        return 0;
    }
    return 1;
}

char* str_of_int(long l) {
    char* str = malloc(12);
    sprintf(str, "%ld", l);
    return str;
}

char* str_con(char* s1, char* s2) {
    char* out = malloc(strlen(s1) + strlen(s2) + 1);
    strcpy(out, s1);
    strcat(out, s2);
    return out;
}
```

stb_image.h

```
/* 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,des
ired_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,des
ired_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,
257,385,513,769,1025,1537,2049,3073,4097,6145,8193,12289,163
85,24577,0,0};

static int stbi__zdist_extra[32] =
{
0,0,0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,1
2,13,13};

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,
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,pr
ior+=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/softimagep
ic/

#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.
------------------------------------------------------------
------------------
*/
```

stb_image_write.h

```
/* stb_image_write - v1.07 - public domain -
http://nothings.org/stb/stb_image_write.h
   writes out PNG/BMP/TGA/JPEG/HDR images to C stdio - Sean
Barrett 2010-2015
                                     no warranty implied;
use at your own risk

   Before #including,

       #define STB_IMAGE_WRITE_IMPLEMENTATION

   in the file that you want to have the implementation.

   Will probably not work correctly with strict-aliasing
optimizations.

ABOUT:

   This header file is a library for writing images to C
stdio. It could be
   adapted to write to memory or a general streaming
interface; let me know.

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

```
   written by a decent optimizing implementation. This
library is designed
   for source code compactness and simplicity, not optimal
image file size
   or run-time performance.

BUILDING:

   You can #define STBIW_ASSERT(x) before the #include to
avoid using assert.h.
   You can #define STBIW_MALLOC(), STBIW_REALLOC(), and
STBIW_FREE() to replace
   malloc,realloc,free.
   You can define STBIW_MEMMOVE() to replace memmove()

USAGE:

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

     int stbi_write_png(char const *filename, int w, int h,
int comp, const void *data, int stride_in_bytes);
     int stbi_write_bmp(char const *filename, int w, int h,
int comp, const void *data);
     int stbi_write_tga(char const *filename, int w, int h,
int comp, const void *data);
     int stbi_write_hdr(char const *filename, int w, int h,
int comp, const float *data);
     int stbi_write_jpg(char const *filename, int w, int h,
int comp, const float *data);

   There are also four equivalent functions that use an
arbitrary write function. You are
   expected to open/close your file-equivalent before and
after calling these:

     int stbi_write_png_to_func(stbi_write_func *func, void
*context, int w, int h, int comp, const void  *data, int
stride_in_bytes);
     int stbi_write_bmp_to_func(stbi_write_func *func, void
*context, int w, int h, int comp, const void  *data);
     int stbi_write_tga_to_func(stbi_write_func *func, void
*context, int w, int h, int comp, const void  *data);
     int stbi_write_hdr_to_func(stbi_write_func *func, void
*context, int w, int h, int comp, const float *data);
     int stbi_write_jpg_to_func(stbi_write_func *func, void
*context, int x, int y, int comp, const void *data, int
quality);

   where the callback is:
      void stbi_write_func(void *context, void *data, int
size);

   You can define STBI_WRITE_NO_STDIO to disable the file
variant of these
   functions, so the library will not use stdio.h at all.
However, this will
```

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

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

   The functions create an image file defined by the
parameters. The image
   is a rectangle of pixels stored from left-to-right, top-
to-bottom.
   Each pixel contains 'comp' channels of data stored
interleaved with 8-bits
   per channel, in the following order: 1=Y, 2=YA, 3=RGB,
4=RGBA. (Y is
   monochrome color.) The rectangle is 'w' pixels wide and
'h' pixels tall.
   The *data pointer points to the first byte of the top-
left-most pixel.
   For PNG, "stride_in_bytes" is the distance in bytes from
the first byte of
   a row of pixels to the first byte of the next row of
pixels.

   PNG creates output files with the same number of
components as the input.
   The BMP format expands Y to RGB in the file format and
does not
   output alpha.

   PNG supports writing rectangles of data even when the
bytes storing rows of
   data are not consecutive in memory (e.g. sub-rectangles
of a larger image),
   by supplying the stride between the beginning of adjacent
rows. The other
   formats do not. (Thus you cannot write a native-format
BMP through the BMP
   writer, both because it is in BGR order and because it
may have padding
   at the end of the line.)

   HDR expects linear float data. Since the format is always
32-bit rgb(e)
   data, alpha (if provided) is discarded, and for
monochrome data it is
   replicated across all three channels.

   TGA supports RLE or non-RLE compressed data. To use non-
RLE-compressed
   data, set the global variable 'stbi_write_tga_with_rle'
to 0.

   JPEG does ignore alpha channels in input data; quality is
between 1 and 100.
   Higher quality looks better but results in a bigger
image.
   JPEG baseline (no JPEG progressive).
```

```
CREDITS:

   PNG/BMP/TGA
      Sean Barrett
   HDR
      Baldur Karlsson
   TGA monochrome:
      Jean-Sebastien Guay
   misc enhancements:
      Tim Kelsey
   TGA RLE
      Alan Hickman
   initial file IO callback implementation
      Emmanuel Julien
   JPEG
      Jon Olick (original jo_jpeg.cpp code)
      Daniel Gibson
   bugfixes:
      github:Chribba
      Guillaume Chereau
      github:jry2
      github:romigrou
      Sergio Gonzalez
      Jonas Karlsson
      Filip Wasil
      Thatcher Ulrich
      github:poppolopoppo
      Patrick Boettcher

LICENSE

  See end of file for license information.

*/

#ifndef INCLUDE_STB_IMAGE_WRITE_H
#define INCLUDE_STB_IMAGE_WRITE_H

#ifdef __cplusplus
extern "C" {
#endif

#ifdef STB_IMAGE_WRITE_STATIC
#define STBIWDEF static
#else
#define STBIWDEF extern
extern int stbi_write_tga_with_rle;
#endif

#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_png(char const *filename, int w, int
h, int comp, const void  *data, int stride_in_bytes);
STBIWDEF int stbi_write_bmp(char const *filename, int w, int
h, int comp, const void  *data);
```

```
STBIWDEF int stbi_write_tga(char const *filename, int w, int
h, int comp, const void  *data);
STBIWDEF int stbi_write_hdr(char const *filename, int w, int
h, int comp, const float *data);
STBIWDEF int stbi_write_jpg(char const *filename, int x, int
y, int comp, const void  *data, int quality);
#endif

typedef void stbi_write_func(void *context, void *data, int
size);

STBIWDEF int stbi_write_png_to_func(stbi_write_func *func,
void *context, int w, int h, int comp, const void  *data,
int stride_in_bytes);
STBIWDEF int stbi_write_bmp_to_func(stbi_write_func *func,
void *context, int w, int h, int comp, const void  *data);
STBIWDEF int stbi_write_tga_to_func(stbi_write_func *func,
void *context, int w, int h, int comp, const void  *data);
STBIWDEF int stbi_write_hdr_to_func(stbi_write_func *func,
void *context, int w, int h, int comp, const float *data);
STBIWDEF int stbi_write_jpg_to_func(stbi_write_func *func,
void *context, int x, int y, int comp, const void  *data,
int quality);

#ifdef __cplusplus
}
#endif

#endif//INCLUDE_STB_IMAGE_WRITE_H

#ifdef STB_IMAGE_WRITE_IMPLEMENTATION

#ifdef _WIN32
   #ifndef _CRT_SECURE_NO_WARNINGS
   #define _CRT_SECURE_NO_WARNINGS
   #endif
   #ifndef _CRT_NONSTDC_NO_DEPRECATE
   #define _CRT_NONSTDC_NO_DEPRECATE
   #endif
#endif

#ifndef STBI_WRITE_NO_STDIO
#include <stdio.h>
#endif // STBI_WRITE_NO_STDIO

#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#if defined(STBIW_MALLOC) && defined(STBIW_FREE) &&
(defined(STBIW_REALLOC) || defined(STBIW_REALLOC_SIZED))
// ok
#elif !defined(STBIW_MALLOC) && !defined(STBIW_FREE) &&
!defined(STBIW_REALLOC) && !defined(STBIW_REALLOC_SIZED)
// ok
```

```
#else
#error "Must define all or none of STBIW_MALLOC, STBIW_FREE,
and STBIW_REALLOC (or STBIW_REALLOC_SIZED)."
#endif

#ifndef STBIW_MALLOC
#define STBIW_MALLOC(sz)        malloc(sz)
#define STBIW_REALLOC(p,newsz)  realloc(p,newsz)
#define STBIW_FREE(p)           free(p)
#endif

#ifndef STBIW_REALLOC_SIZED
#define STBIW_REALLOC_SIZED(p,oldsz,newsz)
STBIW_REALLOC(p,newsz)
#endif


#ifndef STBIW_MEMMOVE
#define STBIW_MEMMOVE(a,b,sz) memmove(a,b,sz)
#endif


#ifndef STBIW_ASSERT
#include <assert.h>
#define STBIW_ASSERT(x) assert(x)
#endif

#define STBIW_UCHAR(x) (unsigned char) ((x) & 0xff)

typedef struct
{
   stbi_write_func *func;
   void *context;
} stbi__write_context;

// initialize a callback-based context
static void stbi__start_write_callbacks(stbi__write_context
*s, stbi_write_func *c, void *context)
{
   s->func    = c;
   s->context = context;
}

#ifndef STBI_WRITE_NO_STDIO

static void stbi__stdio_write(void *context, void *data, int
size)
{
   fwrite(data,1,size,(FILE*) context);
}

static int stbi__start_write_file(stbi__write_context *s,
const char *filename)
{
   FILE *f = fopen(filename, "wb");
```

```
   stbi__start_write_callbacks(s, stbi__stdio_write, (void
*) f);
   return f != NULL;
}

static void stbi__end_write_file(stbi__write_context *s)
{
   fclose((FILE *)s->context);
}

#endif // !STBI_WRITE_NO_STDIO

typedef unsigned int stbiw_uint32;
typedef int stb_image_write_test[sizeof(stbiw_uint32)==4 ? 1
: -1];

#ifdef STB_IMAGE_WRITE_STATIC
static int stbi_write_tga_with_rle = 1;
#else
int stbi_write_tga_with_rle = 1;
#endif

static void stbiw__writefv(stbi__write_context *s, const
char *fmt, va_list v)
{
   while (*fmt) {
      switch (*fmt++) {
         case ' ': break;
         case '1': { unsigned char x = STBIW_UCHAR(va_arg(v,
int));
                     s->func(s->context,&x,1);
                     break; }
         case '2': { int x = va_arg(v,int);
                     unsigned char b[2];
                     b[0] = STBIW_UCHAR(x);
                     b[1] = STBIW_UCHAR(x>>8);
                     s->func(s->context,b,2);
                     break; }
         case '4': { stbiw_uint32 x = va_arg(v,int);
                     unsigned char b[4];
                     b[0]=STBIW_UCHAR(x);
                     b[1]=STBIW_UCHAR(x>>8);
                     b[2]=STBIW_UCHAR(x>>16);
                     b[3]=STBIW_UCHAR(x>>24);
                     s->func(s->context,b,4);
                     break; }
         default:
            STBIW_ASSERT(0);
            return;
      }
   }
}

static void stbiw__writef(stbi__write_context *s, const char
*fmt, ...)
{
```

```
   va_list v;
   va_start(v, fmt);
   stbiw__writefv(s, fmt, v);
   va_end(v);
}

static void stbiw__putc(stbi__write_context *s, unsigned
char c)
{
   s->func(s->context, &c, 1);
}

static void stbiw__write3(stbi__write_context *s, unsigned
char a, unsigned char b, unsigned char c)
{
   unsigned char arr[3];
   arr[0] = a, arr[1] = b, arr[2] = c;
   s->func(s->context, arr, 3);
}

static void stbiw__write_pixel(stbi__write_context *s, int
rgb_dir, int comp, int write_alpha, int expand_mono,
unsigned char *d)
{
   unsigned char bg[3] = { 255, 0, 255}, px[3];
   int k;

   if (write_alpha < 0)
      s->func(s->context, &d[comp - 1], 1);

   switch (comp) {
      case 2: // 2 pixels = mono + alpha, alpha is written
separately, so same as 1-channel case
      case 1:
         if (expand_mono)
            stbiw__write3(s, d[0], d[0], d[0]); //
monochrome bmp
         else
            s->func(s->context, d, 1);  // monochrome TGA
         break;
      case 4:
         if (!write_alpha) {
            // composite against pink background
            for (k = 0; k < 3; ++k)
               px[k] = bg[k] + ((d[k] - bg[k]) * d[3]) /
255;
            stbiw__write3(s, px[1 - rgb_dir], px[1], px[1 +
rgb_dir]);
            break;
         }
         /* FALLTHROUGH */
      case 3:
         stbiw__write3(s, d[1 - rgb_dir], d[1], d[1 +
rgb_dir]);
         break;
   }
```

```
   if (write_alpha > 0)
      s->func(s->context, &d[comp - 1], 1);
}

static void stbiw__write_pixels(stbi__write_context *s, int
rgb_dir, int vdir, int x, int y, int comp, void *data, int
write_alpha, int scanline_pad, int expand_mono)
{
   stbiw_uint32 zero = 0;
   int i,j, j_end;

   if (y <= 0)
      return;

   if (vdir < 0)
      j_end = -1, j = y-1;
   else
      j_end =  y, j = 0;

   for (; j != j_end; j += vdir) {
      for (i=0; i < x; ++i) {
         unsigned char *d = (unsigned char *) data +
(j*x+i)*comp;
         stbiw__write_pixel(s, rgb_dir, comp, write_alpha,
expand_mono, d);
      }
      s->func(s->context, &zero, scanline_pad);
   }
}

static int stbiw__outfile(stbi__write_context *s, int
rgb_dir, int vdir, int x, int y, int comp, int expand_mono,
void *data, int alpha, int pad, const char *fmt, ...)
{
   if (y < 0 || x < 0) {
      return 0;
   } else {
      va_list v;
      va_start(v, fmt);
      stbiw__writefv(s, fmt, v);
      va_end(v);

stbiw__write_pixels(s,rgb_dir,vdir,x,y,comp,data,alpha,pad,
expand_mono);
      return 1;
   }
}

static int stbi_write_bmp_core(stbi__write_context *s, int
x, int y, int comp, const void *data)
{
   int pad = (-x*3) & 3;
   return stbiw__outfile(s,-1,-1,x,y,comp,1,(void *)
data,0,pad,
           "11 4 22 4" "4 44 22 444444",
```

```
           'B', 'M', 14+40+(x*3+pad)*y, 0,0, 14+40,  // file
header
            40, x,y, 1,24, 0,0,0,0,0,0);             //
bitmap header
}

STBIWDEF int stbi_write_bmp_to_func(stbi_write_func *func,
void *context, int x, int y, int comp, const void *data)
{
   stbi__write_context s;
   stbi__start_write_callbacks(&s, func, context);
   return stbi_write_bmp_core(&s, x, y, comp, data);
}

#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_bmp(char const *filename, int x, int
y, int comp, const void *data)
{
   stbi__write_context s;
   if (stbi__start_write_file(&s,filename)) {
      int r = stbi_write_bmp_core(&s, x, y, comp, data);
      stbi__end_write_file(&s);
      return r;
   } else
      return 0;
}
#endif //!STBI_WRITE_NO_STDIO

static int stbi_write_tga_core(stbi__write_context *s, int
x, int y, int comp, void *data)
{
   int has_alpha = (comp == 2 || comp == 4);
   int colorbytes = has_alpha ? comp-1 : comp;
   int format = colorbytes < 2 ? 3 : 2; // 3 color channels
(RGB/RGBA) = 2, 1 color channel (Y/YA) = 3

   if (y < 0 || x < 0)
      return 0;

   if (!stbi_write_tga_with_rle) {
      return stbiw__outfile(s, -1, -1, x, y, comp, 0, (void
*) data, has_alpha, 0,
         "111 221 2222 11", 0, 0, format, 0, 0, 0, 0, 0, x,
y, (colorbytes + has_alpha) * 8, has_alpha * 8);
   } else {
      int i,j,k;

      stbiw__writef(s, "111 221 2222 11", 0,0,format+8,
0,0,0, 0,0,x,y, (colorbytes + has_alpha) * 8, has_alpha *
8);

      for (j = y - 1; j >= 0; --j) {
          unsigned char *row = (unsigned char *) data + j *
x * comp;
         int len;
```

```
         for (i = 0; i < x; i += len) {
            unsigned char *begin = row + i * comp;
            int diff = 1;
            len = 1;

            if (i < x - 1) {
               ++len;
               diff = memcmp(begin, row + (i + 1) * comp,
comp);
               if (diff) {
                  const unsigned char *prev = begin;
                  for (k = i + 2; k < x && len < 128; ++k) {
                     if (memcmp(prev, row + k * comp, comp))
{
                        prev += comp;
                        ++len;
                     } else {
                        --len;
                        break;
                     }
                  }
               } else {
                  for (k = i + 2; k < x && len < 128; ++k) {
                     if (!memcmp(begin, row + k * comp,
comp)) {
                        ++len;
                     } else {
                        break;
                     }
                  }
               }
            }

            if (diff) {
               unsigned char header = STBIW_UCHAR(len - 1);
               s->func(s->context, &header, 1);
               for (k = 0; k < len; ++k) {
                  stbiw__write_pixel(s, -1, comp, has_alpha,
0, begin + k * comp);
               }
            } else {
               unsigned char header = STBIW_UCHAR(len -
129);
               s->func(s->context, &header, 1);
               stbiw__write_pixel(s, -1, comp, has_alpha, 0,
begin);
            }
         }
      }
   }
   return 1;
}

STBIWDEF int stbi_write_tga_to_func(stbi_write_func *func,
void *context, int x, int y, int comp, const void *data)
{
```

```
   stbi__write_context s;
   stbi__start_write_callbacks(&s, func, context);
   return stbi_write_tga_core(&s, x, y, comp, (void *)
data);
}

#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_tga(char const *filename, int x, int
y, int comp, const void *data)
{
   stbi__write_context s;
   if (stbi__start_write_file(&s,filename)) {
      int r = stbi_write_tga_core(&s, x, y, comp, (void *)
data);
      stbi__end_write_file(&s);
      return r;
   } else
      return 0;
}
#endif

//
************************************************************
************************************
// Radiance RGBE HDR writer
// by Baldur Karlsson

#define stbiw__max(a, b)  ((a) > (b) ? (a) : (b))

void stbiw__linear_to_rgbe(unsigned char *rgbe, float
*linear)
{
   int exponent;
   float maxcomp = stbiw__max(linear[0],
stbiw__max(linear[1], linear[2]));

   if (maxcomp < 1e-32f) {
      rgbe[0] = rgbe[1] = rgbe[2] = rgbe[3] = 0;
   } else {
      float normalize = (float) frexp(maxcomp, &exponent) *
256.0f/maxcomp;

      rgbe[0] = (unsigned char)(linear[0] * normalize);
      rgbe[1] = (unsigned char)(linear[1] * normalize);
      rgbe[2] = (unsigned char)(linear[2] * normalize);
      rgbe[3] = (unsigned char)(exponent + 128);
   }
}

void stbiw__write_run_data(stbi__write_context *s, int
length, unsigned char databyte)
{
   unsigned char lengthbyte = STBIW_UCHAR(length+128);
   STBIW_ASSERT(length+128 <= 255);
   s->func(s->context, &lengthbyte, 1);
   s->func(s->context, &databyte, 1);
```

```
}

void stbiw__write_dump_data(stbi__write_context *s, int
length, unsigned char *data)
{
   unsigned char lengthbyte = STBIW_UCHAR(length);
   STBIW_ASSERT(length <= 128); // inconsistent with spec
but consistent with official code
   s->func(s->context, &lengthbyte, 1);
   s->func(s->context, data, length);
}

void stbiw__write_hdr_scanline(stbi__write_context *s, int
width, int ncomp, unsigned char *scratch, float *scanline)
{
   unsigned char scanlineheader[4] = { 2, 2, 0, 0 };
   unsigned char rgbe[4];
   float linear[3];
   int x;

   scanlineheader[2] = (width&0xff00)>>8;
   scanlineheader[3] = (width&0x00ff);

   /* skip RLE for images too small or large */
   if (width < 8 || width >= 32768) {
      for (x=0; x < width; x++) {
         switch (ncomp) {
            case 4: /* fallthrough */
            case 3: linear[2] = scanline[x*ncomp + 2];
                    linear[1] = scanline[x*ncomp + 1];
                    linear[0] = scanline[x*ncomp + 0];
                    break;
            default:
                    linear[0] = linear[1] = linear[2] =
scanline[x*ncomp + 0];
                    break;
         }
         stbiw__linear_to_rgbe(rgbe, linear);
         s->func(s->context, rgbe, 4);
      }
   } else {
      int c,r;
      /* encode into scratch buffer */
      for (x=0; x < width; x++) {
         switch(ncomp) {
            case 4: /* fallthrough */
            case 3: linear[2] = scanline[x*ncomp + 2];
                    linear[1] = scanline[x*ncomp + 1];
                    linear[0] = scanline[x*ncomp + 0];
                    break;
            default:
                    linear[0] = linear[1] = linear[2] =
scanline[x*ncomp + 0];
                    break;
         }
         stbiw__linear_to_rgbe(rgbe, linear);
```

```
         scratch[x + width*0] = rgbe[0];
         scratch[x + width*1] = rgbe[1];
         scratch[x + width*2] = rgbe[2];
         scratch[x + width*3] = rgbe[3];
      }

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

      /* RLE each component separately */
      for (c=0; c < 4; c++) {
         unsigned char *comp = &scratch[width*c];

         x = 0;
         while (x < width) {
            // find first run
            r = x;
            while (r+2 < width) {
               if (comp[r] == comp[r+1] && comp[r] ==
comp[r+2])
                  break;
               ++r;
            }
            if (r+2 >= width)
               r = width;
            // dump up to first run
            while (x < r) {
               int len = r-x;
               if (len > 128) len = 128;
               stbiw__write_dump_data(s, len, &comp[x]);
               x += len;
            }
            // if there's a run, output it
            if (r+2 < width) { // same test as what we break
out of in search loop, so only true if we break'd
               // find next byte after run
               while (r < width && comp[r] == comp[x])
                  ++r;
               // output run up to r
               while (x < r) {
                  int len = r-x;
                  if (len > 127) len = 127;
                  stbiw__write_run_data(s, len, comp[x]);
                  x += len;
               }
            }
         }
      }
   }
}

static int stbi_write_hdr_core(stbi__write_context *s, int
x, int y, int comp, float *data)
{
   if (y <= 0 || x <= 0 || data == NULL)
      return 0;
   else {
```

```
      // Each component is stored separately. Allocate
scratch space for full output scanline.
      unsigned char *scratch = (unsigned char *)
STBIW_MALLOC(x*4);
      int i, len;
      char buffer[128];
      char header[] = "#?RADIANCE\n# Written by
stb_image_write.h\nFORMAT=32-bit_rle_rgbe\n";
      s->func(s->context, header, sizeof(header)-1);

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

      for(i=0; i < y; i++)
         stbiw__write_hdr_scanline(s, x, comp, scratch, data
+ comp*i*x);
      STBIW_FREE(scratch);
      return 1;
   }
}

STBIWDEF int stbi_write_hdr_to_func(stbi_write_func *func,
void *context, int x, int y, int comp, const float *data)
{
   stbi__write_context s;
   stbi__start_write_callbacks(&s, func, context);
   return stbi_write_hdr_core(&s, x, y, comp, (float *)
data);
}

#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_hdr(char const *filename, int x, int
y, int comp, const float *data)
{
   stbi__write_context s;
   if (stbi__start_write_file(&s,filename)) {
      int r = stbi_write_hdr_core(&s, x, y, comp, (float *)
data);
      stbi__end_write_file(&s);
      return r;
   } else
      return 0;
}
#endif // STBI_WRITE_NO_STDIO


////////////////////////////////////////////////////////////
/////////////////
//
// PNG writer
//

// stretchy buffer; stbiw__sbpush() == vector<>::push_back()
-- stbiw__sbcount() == vector<>::size()
#define stbiw__sbraw(a) ((int *) (a) - 2)
```

```
#define stbiw__sbm(a)   stbiw__sbraw(a)[0]
#define stbiw__sbn(a)   stbiw__sbraw(a)[1]

#define stbiw__sbneedgrow(a,n)  ((a)==0 || stbiw__sbn(a)+n
>= stbiw__sbm(a))
#define stbiw__sbmaybegrow(a,n) (stbiw__sbneedgrow(a,(n)) ?
stbiw__sbgrow(a,n) : 0)
#define stbiw__sbgrow(a,n)  stbiw__sbgrowf((void **) &(a),
(n), sizeof(*(a)))

#define stbiw__sbpush(a, v)      (stbiw__sbmaybegrow(a,1),
(a)[stbiw__sbn(a)++] = (v))
#define stbiw__sbcount(a)        ((a) ? stbiw__sbn(a) : 0)
#define stbiw__sbfree(a)         ((a) ?
STBIW_FREE(stbiw__sbraw(a)),0 : 0)

static void *stbiw__sbgrowf(void **arr, int increment, int
itemsize)
{
   int m = *arr ? 2*stbiw__sbm(*arr)+increment :
increment+1;
   void *p = STBIW_REALLOC_SIZED(*arr ? stbiw__sbraw(*arr) :
0, *arr ? (stbiw__sbm(*arr)*itemsize + sizeof(int)*2) : 0,
itemsize * m + sizeof(int)*2);
   STBIW_ASSERT(p);
   if (p) {
      if (!*arr) ((int *) p)[1] = 0;
      *arr = (void *) ((int *) p + 2);
      stbiw__sbm(*arr) = m;
   }
   return *arr;
}

static unsigned char *stbiw__zlib_flushf(unsigned char
*data, unsigned int *bitbuffer, int *bitcount)
{
   while (*bitcount >= 8) {
      stbiw__sbpush(data, STBIW_UCHAR(*bitbuffer));
      *bitbuffer >>= 8;
      *bitcount -= 8;
   }
   return data;
}

static int stbiw__zlib_bitrev(int code, int codebits)
{
   int res=0;
   while (codebits--) {
      res = (res << 1) | (code & 1);
      code >>= 1;
   }
   return res;
}

static unsigned int stbiw__zlib_countm(unsigned char *a,
unsigned char *b, int limit)
```

```
{
   int i;
   for (i=0; i < limit && i < 258; ++i)
      if (a[i] != b[i]) break;
   return i;
}

static unsigned int stbiw__zhash(unsigned char *data)
{
   stbiw_uint32 hash = data[0] + (data[1] << 8) + (data[2]
<< 16);
   hash ^= hash << 3;
   hash += hash >> 5;
   hash ^= hash << 4;
   hash += hash >> 17;
   hash ^= hash << 25;
   hash += hash >> 6;
   return hash;
}

#define stbiw__zlib_flush() (out = stbiw__zlib_flushf(out,
&bitbuf, &bitcount))
#define stbiw__zlib_add(code,codebits) \
      (bitbuf |= (code) << bitcount, bitcount += (codebits),
stbiw__zlib_flush())
#define stbiw__zlib_huffa(b,c)
stbiw__zlib_add(stbiw__zlib_bitrev(b,c),c)
// default huffman tables
#define stbiw__zlib_huff1(n)  stbiw__zlib_huffa(0x30 + (n),
8)
#define stbiw__zlib_huff2(n)  stbiw__zlib_huffa(0x190 + (n)-
144, 9)
#define stbiw__zlib_huff3(n)  stbiw__zlib_huffa(0 + (n)-
256,7)
#define stbiw__zlib_huff4(n)  stbiw__zlib_huffa(0xc0 + (n)-
280,8)
#define stbiw__zlib_huff(n)  ((n) <= 143 ?
stbiw__zlib_huff1(n) : (n) <= 255 ? stbiw__zlib_huff2(n) :
(n) <= 279 ? stbiw__zlib_huff3(n) : stbiw__zlib_huff4(n))
#define stbiw__zlib_huffb(n) ((n) <= 143 ?
stbiw__zlib_huff1(n) : stbiw__zlib_huff2(n))

#define stbiw__ZHASH   16384

unsigned char * stbi_zlib_compress(unsigned char *data, int
data_len, int *out_len, int quality)
{
   static unsigned short lengthc[] = {
3,4,5,6,7,8,9,10,11,13,15,17,19,23,27,31,35,43,51,59,67,83,9
9,115,131,163,195,227,258, 259 };
   static unsigned char  lengtheb[]= { 0,0,0,0,0,0,0, 0, 1,
1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4,  4,  5,  5,  5,
5,  0 };
   static unsigned short distc[]   = {
1,2,3,4,5,7,9,13,17,25,33,49,65,97,129,193,257,385,513,769,1
```

```
025,1537,2049,3073,4097,6145,8193,12289,16385,24577, 32768
};
   static unsigned char  disteb[]  = {
0,0,0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,1
2,13,13 };
   unsigned int bitbuf=0;
   int i,j, bitcount=0;
   unsigned char *out = NULL;
   unsigned char ***hash_table = (unsigned char***)
STBIW_MALLOC(stbiw__ZHASH * sizeof(char**));
   if (quality < 5) quality = 5;

   stbiw__sbpush(out, 0x78);   // DEFLATE 32K window
   stbiw__sbpush(out, 0x5e);   // FLEVEL = 1
   stbiw__zlib_add(1,1);  // BFINAL = 1
   stbiw__zlib_add(1,2);  // BTYPE = 1 -- fixed huffman

   for (i=0; i < stbiw__ZHASH; ++i)
      hash_table[i] = NULL;

   i=0;
   while (i < data_len-3) {
      // hash next 3 bytes of data to be compressed
      int h = stbiw__zhash(data+i)&(stbiw__ZHASH-1), best=3;
      unsigned char *bestloc = 0;
      unsigned char **hlist = hash_table[h];
      int n = stbiw__sbcount(hlist);
      for (j=0; j < n; ++j) {
         if (hlist[j]-data > i-32768) { // if entry lies
within window
            int d = stbiw__zlib_countm(hlist[j], data+i,
data_len-i);
            if (d >= best) best=d,bestloc=hlist[j];
         }
      }
      // when hash table entry is too long, delete half the
entries
      if (hash_table[h] && stbiw__sbn(hash_table[h]) ==
2*quality) {
         STBIW_MEMMOVE(hash_table[h], hash_table[h]+quality,
sizeof(hash_table[h][0])*quality);
         stbiw__sbn(hash_table[h]) = quality;
      }
      stbiw__sbpush(hash_table[h],data+i);

      if (bestloc) {
         // "lazy matching" - check match at *next* byte,
and if it's better, do cur byte as literal
         h = stbiw__zhash(data+i+1)&(stbiw__ZHASH-1);
         hlist = hash_table[h];
         n = stbiw__sbcount(hlist);
         for (j=0; j < n; ++j) {
            if (hlist[j]-data > i-32767) {
               int e = stbiw__zlib_countm(hlist[j],
data+i+1, data_len-i-1);
```

```
                if (e > best) { // if next match is better,
bail on current match
                   bestloc = NULL;
                   break;
                }
             }
          }
       }

       if (bestloc) {
          int d = (int) (data+i - bestloc); // distance back
          STBIW_ASSERT(d <= 32767 && best <= 258);
          for (j=0; best > lengthc[j+1]-1; ++j);
          stbiw__zlib_huff(j+257);
          if (lengtheb[j]) stbiw__zlib_add(best - lengthc[j],
lengtheb[j]);
          for (j=0; d > distc[j+1]-1; ++j);
          stbiw__zlib_add(stbiw__zlib_bitrev(j,5),5);
          if (disteb[j]) stbiw__zlib_add(d - distc[j],
disteb[j]);
          i += best;
       } else {
          stbiw__zlib_huffb(data[i]);
          ++i;
       }
    }
    // write out final bytes
    for (;i < data_len; ++i)
       stbiw__zlib_huffb(data[i]);
    stbiw__zlib_huff(256); // end of block
    // pad with 0 bits to byte boundary
    while (bitcount)
       stbiw__zlib_add(0,1);

    for (i=0; i < stbiw__ZHASH; ++i)
       (void) stbiw__sbfree(hash_table[i]);
    STBIW_FREE(hash_table);

    {
       // compute adler32 on input
       unsigned int s1=1, s2=0;
       int blocklen = (int) (data_len % 5552);
       j=0;
       while (j < data_len) {
          for (i=0; i < blocklen; ++i) s1 += data[j+i], s2 +=
s1;
          s1 %= 65521, s2 %= 65521;
          j += blocklen;
          blocklen = 5552;
       }
       stbiw__sbpush(out, STBIW_UCHAR(s2 >> 8));
       stbiw__sbpush(out, STBIW_UCHAR(s2));
       stbiw__sbpush(out, STBIW_UCHAR(s1 >> 8));
       stbiw__sbpush(out, STBIW_UCHAR(s1));
    }
    *out_len = stbiw__sbn(out);
```

```
   // make returned pointer freeable
   STBIW_MEMMOVE(stbiw__sbraw(out), out, *out_len);
   return (unsigned char *) stbiw__sbraw(out);
}

static unsigned int stbiw__crc32(unsigned char *buffer, int
len)
{
   static unsigned int crc_table[256] =
   {
      0x00000000, 0x77073096, 0xEE0E612C, 0x990951BA,
0x076DC419, 0x706AF48F, 0xE963A535, 0x9E6495A3,
      0x0eDB8832, 0x79DCB8A4, 0xE0D5E91E, 0x97D2D988,
0x09B64C2B, 0x7EB17CBD, 0xE7B82D07, 0x90BF1D91,
      0x1DB71064, 0x6AB020F2, 0xF3B97148, 0x84BE41DE,
0x1ADAD47D, 0x6DDDE4EB, 0xF4D4B551, 0x83D385C7,
      0x136C9856, 0x646BA8C0, 0xFD62F97A, 0x8A65C9EC,
0x14015C4F, 0x63066CD9, 0xFA0F3D63, 0x8D080DF5,
      0x3B6E20C8, 0x4C69105E, 0xD56041E4, 0xA2677172,
0x3C03E4D1, 0x4B04D447, 0xD20D85FD, 0xA50AB56B,
      0x35B5A8FA, 0x42B2986C, 0xDBBBC9D6, 0xACBCF940,
0x32D86CE3, 0x45DF5C75, 0xDCD60DCF, 0xABD13D59,
      0x26D930AC, 0x51DE003A, 0xC8D75180, 0xBFD06116,
0x21B4F4B5, 0x56B3C423, 0xCFBA9599, 0xB8BDA50F,
      0x2802B89E, 0x5F058808, 0xC60CD9B2, 0xB10BE924,
0x2F6F7C87, 0x58684C11, 0xC1611DAB, 0xB6662D3D,
      0x76DC4190, 0x01DB7106, 0x98D220BC, 0xEFD5102A,
0x71B18589, 0x06B6B51F, 0x9FBFE4A5, 0xE8B8D433,
      0x7807C9A2, 0x0F00F934, 0x9609A88E, 0xE10E9818,
0x7F6A0DBB, 0x086D3D2D, 0x91646C97, 0xE6635C01,
      0x6B6B51F4, 0x1C6C6162, 0x856530D8, 0xF262004E,
0x6C0695ED, 0x1B01A57B, 0x8208F4C1, 0xF50FC457,
      0x65B0D9C6, 0x12B7E950, 0x8BBEB8EA, 0xFCB9887C,
0x62DD1DDF, 0x15DA2D49, 0x8CD37CF3, 0xFBD44C65,
      0x4DB26158, 0x3AB551CE, 0xA3BC0074, 0xD4BB30E2,
0x4ADFA541, 0x3DD895D7, 0xA4D1C46D, 0xD3D6F4FB,
      0x4369E96A, 0x346ED9FC, 0xAD678846, 0xDA60B8D0,
0x44042D73, 0x33031DE5, 0xAA0A4C5F, 0xDD0D7CC9,
      0x5005713C, 0x270241AA, 0xBE0B1010, 0xC90C2086,
0x5768B525, 0x206F85B3, 0xB966D409, 0xCE61E49F,
      0x5EDEF90E, 0x29D9C998, 0xB0D09822, 0xC7D7A8B4,
0x59B33D17, 0x2EB40D81, 0xB7BD5C3B, 0xC0BA6CAD,
      0xEDB88320, 0x9ABFB3B6, 0x03B6E20C, 0x74B1D29A,
0xEAD54739, 0x9DD277AF, 0x04DB2615, 0x73DC1683,
      0xE3630B12, 0x94643B84, 0x0D6D6A3E, 0x7A6A5AA8,
0xE40ECF0B, 0x9309FF9D, 0x0A00AE27, 0x7D079EB1,
      0xF00F9344, 0x8708A3D2, 0x1E01F268, 0x6906C2FE,
0xF762575D, 0x806567CB, 0x196C3671, 0x6E6B06E7,
      0xFED41B76, 0x89D32BE0, 0x10DA7A5A, 0x67DD4ACC,
0xF9B9DF6F, 0x8EBEEFF9, 0x17B7BE43, 0x60B08ED5,
      0xD6D6A3E8, 0xA1D1937E, 0x38D8C2C4, 0x4FDFF252,
0xD1BB67F1, 0xA6BC5767, 0x3FB506DD, 0x48B2364B,
      0xD80D2BDA, 0xAF0A1B4C, 0x36034AF6, 0x41047A60,
0xDF60EFC3, 0xA867DF55, 0x316E8EEF, 0x4669BE79,
      0xCB61B38C, 0xBC66831A, 0x256FD2A0, 0x5268E236,
0xCC0C7795, 0xBB0B4703, 0x220216B9, 0x5505262F,
```

```
      0xC5BA3BBE, 0xB2BD0B28, 0x2BB45A92, 0x5CB36A04,
0xC2D7FFA7, 0xB5D0CF31, 0x2CD99E8B, 0x5BDEAE1D,
      0x9B64C2B0, 0xEC63F226, 0x756AA39C, 0x026D930A,
0x9C0906A9, 0xEB0E363F, 0x72076785, 0x05005713,
      0x95BF4A82, 0xE2B87A14, 0x7BB12BAE, 0x0CB61B38,
0x92D28E9B, 0xE5D5BE0D, 0x7CDCEFB7, 0x0BDBDF21,
      0x86D3D2D4, 0xF1D4E242, 0x68DDB3F8, 0x1FDA836E,
0x81BE16CD, 0xF6B9265B, 0x6FB077E1, 0x18B74777,
      0x88085AE6, 0xFF0F6A70, 0x66063BCA, 0x11010B5C,
0x8F659EFF, 0xF862AE69, 0x616BFFD3, 0x166CCF45,
      0xA00AE278, 0xD70DD2EE, 0x4E048354, 0x3903B3C2,
0xA7672661, 0xD06016F7, 0x4969474D, 0x3E6E77DB,
      0xAED16A4A, 0xD9D65ADC, 0x40DF0B66, 0x37D83BF0,
0xA9BCAE53, 0xDEBB9EC5, 0x47B2CF7F, 0x30B5FFE9,
      0xBDBDF21C, 0xCABAC28A, 0x53B39330, 0x24B4A3A6,
0xBAD03605, 0xCDD70693, 0x54DE5729, 0x23D967BF,
      0xB3667A2E, 0xC4614AB8, 0x5D681B02, 0x2A6F2B94,
0xB40BBE37, 0xC30C8EA1, 0x5A05DF1B, 0x2D02EF8D
   };

   unsigned int crc = ~0u;
   int i;
   for (i=0; i < len; ++i)
      crc = (crc >> 8) ^ crc_table[buffer[i] ^ (crc &
0xff)];
   return ~crc;
}

#define stbiw__wpng4(o,a,b,c,d)
((o)[0]=STBIW_UCHAR(a),(o)[1]=STBIW_UCHAR(b),(o)[2]=STBIW_UC
HAR(c),(o)[3]=STBIW_UCHAR(d),(o)+=4)
#define stbiw__wp32(data,v) stbiw__wpng4(data,
(v)>>24,(v)>>16,(v)>>8,(v));
#define stbiw__wptag(data,s) stbiw__wpng4(data,
s[0],s[1],s[2],s[3])

static void stbiw__wpcrc(unsigned char **data, int len)
{
   unsigned int crc = stbiw__crc32(*data - len - 4, len+4);
   stbiw__wp32(*data, crc);
}

static unsigned char stbiw__paeth(int a, int b, int c)
{
   int p = a + b - c, pa = abs(p-a), pb = abs(p-b), pc =
abs(p-c);
   if (pa <= pb && pa <= pc) return STBIW_UCHAR(a);
   if (pb <= pc) return STBIW_UCHAR(b);
   return STBIW_UCHAR(c);
}

// @OPTIMIZE: provide an option that always forces left-
predict or paeth predict
unsigned char *stbi_write_png_to_mem(unsigned char *pixels,
int stride_bytes, int x, int y, int n, int *out_len)
{
```

```
   int ctype[5] = { -1, 0, 4, 2, 6 };
   unsigned char sig[8] = { 137,80,78,71,13,10,26,10 };
   unsigned char *out,*o, *filt, *zlib;
   signed char *line_buffer;
   int i,j,k,p,zlen;

   if (stride_bytes == 0)
      stride_bytes = x * n;

   filt = (unsigned char *) STBIW_MALLOC((x*n+1) * y); if
(!filt) return 0;
   line_buffer = (signed char *) STBIW_MALLOC(x * n); if
(!line_buffer) { STBIW_FREE(filt); return 0; }
   for (j=0; j < y; ++j) {
      static int mapping[] = { 0,1,2,3,4 };
      static int firstmap[] = { 0,1,0,5,6 };
      int *mymap = (j != 0) ? mapping : firstmap;
      int best = 0, bestval = 0x7fffffff;
      for (p=0; p < 2; ++p) {
         for (k= p?best:0; k < 5; ++k) { // @TODO: clarity:
rewrite this to go 0..5, and 'continue' the unwanted ones
during 2nd pass
            int type = mymap[k],est=0;
            unsigned char *z = pixels + stride_bytes*j;
            for (i=0; i < n; ++i)
               switch (type) {
                  case 0: line_buffer[i] = z[i]; break;
                  case 1: line_buffer[i] = z[i]; break;
                  case 2: line_buffer[i] = z[i] - z[i-
stride_bytes]; break;
                  case 3: line_buffer[i] = z[i] - (z[i-
stride_bytes]>>1); break;
                  case 4: line_buffer[i] = (signed char)
(z[i] - stbiw__paeth(0,z[i-stride_bytes],0)); break;
                  case 5: line_buffer[i] = z[i]; break;
                  case 6: line_buffer[i] = z[i]; break;
               }
            for (i=n; i < x*n; ++i) {
               switch (type) {
                  case 0: line_buffer[i] = z[i]; break;
                  case 1: line_buffer[i] = z[i] - z[i-n];
break;
                  case 2: line_buffer[i] = z[i] - z[i-
stride_bytes]; break;
                  case 3: line_buffer[i] = z[i] - ((z[i-n] +
z[i-stride_bytes])>>1); break;
                  case 4: line_buffer[i] = z[i] -
stbiw__paeth(z[i-n], z[i-stride_bytes], z[i-stride_bytes-
n]); break;
                  case 5: line_buffer[i] = z[i] - (z[i-
n]>>1); break;
                  case 6: line_buffer[i] = z[i] -
stbiw__paeth(z[i-n], 0,0); break;
               }
            }
            if (p) break;
```

```
            for (i=0; i < x*n; ++i)
               est += abs((signed char) line_buffer[i]);
            if (est < bestval) { bestval = est; best = k; }
         }
      }
      // when we get here, best contains the filter type,
and line_buffer contains the data
      filt[j*(x*n+1)] = (unsigned char) best;
      STBIW_MEMMOVE(filt+j*(x*n+1)+1, line_buffer, x*n);
   }
   STBIW_FREE(line_buffer);
   zlib = stbi_zlib_compress(filt, y*( x*n+1), &zlen, 8); //
increase 8 to get smaller but use more memory
   STBIW_FREE(filt);
   if (!zlib) return 0;

   // each tag requires 12 bytes of overhead
   out = (unsigned char *) STBIW_MALLOC(8 + 12+13 + 12+zlen
+ 12);
   if (!out) return 0;
   *out_len = 8 + 12+13 + 12+zlen + 12;

   o=out;
   STBIW_MEMMOVE(o,sig,8); o+= 8;
   stbiw__wp32(o, 13); // header length
   stbiw__wptag(o, "IHDR");
   stbiw__wp32(o, x);
   stbiw__wp32(o, y);
   *o++ = 8;
   *o++ = STBIW_UCHAR(ctype[n]);
   *o++ = 0;
   *o++ = 0;
   *o++ = 0;
   stbiw__wpcrc(&o,13);

   stbiw__wp32(o, zlen);
   stbiw__wptag(o, "IDAT");
   STBIW_MEMMOVE(o, zlib, zlen);
   o += zlen;
   STBIW_FREE(zlib);
   stbiw__wpcrc(&o, zlen);

   stbiw__wp32(o,0);
   stbiw__wptag(o, "IEND");
   stbiw__wpcrc(&o,0);

   STBIW_ASSERT(o == out + *out_len);

   return out;
}

#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_png(char const *filename, int x, int
y, int comp, const void *data, int stride_bytes)
{
   FILE *f;
```

```
   int len;
   unsigned char *png = stbi_write_png_to_mem((unsigned char
*) data, stride_bytes, x, y, comp, &len);
   if (png == NULL) return 0;
   f = fopen(filename, "wb");
   if (!f) { STBIW_FREE(png); return 0; }
   fwrite(png, 1, len, f);
   fclose(f);
   STBIW_FREE(png);
   return 1;
}
#endif

STBIWDEF int stbi_write_png_to_func(stbi_write_func *func,
void *context, int x, int y, int comp, const void *data, int
stride_bytes)
{
   int len;
   unsigned char *png = stbi_write_png_to_mem((unsigned char
*) data, stride_bytes, x, y, comp, &len);
   if (png == NULL) return 0;
   func(context, png, len);
   STBIW_FREE(png);
   return 1;
}


/*
************************************************************
***************
 *
 * JPEG writer
 *
 * This is based on Jon Olick's jo_jpeg.cpp:
 * public domain Simple, Minimalistic JPEG writer -
http://www.jonolick.com/code.html
 */

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

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

static void stbiw__jpg_writeBits(stbi__write_context *s, int
*bitBufP, int *bitCntP, const unsigned short *bs) {
   int bitBuf = *bitBufP, bitCnt = *bitCntP;
   bitCnt += bs[1];
   bitBuf |= bs[0] << (24 - bitCnt);
   while(bitCnt >= 8) {
      unsigned char c = (bitBuf >> 16) & 255;
      stbiw__putc(s, c);
      if(c == 255) {
         stbiw__putc(s, 0);
      }
```

```
      bitBuf <<= 8;
      bitCnt -= 8;
   }
   *bitBufP = bitBuf;
   *bitCntP = bitCnt;
}

static void stbiw__jpg_DCT(float *d0p, float *d1p, float
*d2p, float *d3p, float *d4p, float *d5p, float *d6p, float
*d7p) {
   float d0 = *d0p, d1 = *d1p, d2 = *d2p, d3 = *d3p, d4 =
*d4p, d5 = *d5p, d6 = *d6p, d7 = *d7p;
   float z1, z2, z3, z4, z5, z11, z13;

   float tmp0 = d0 + d7;
   float tmp7 = d0 - d7;
   float tmp1 = d1 + d6;
   float tmp6 = d1 - d6;
   float tmp2 = d2 + d5;
   float tmp5 = d2 - d5;
   float tmp3 = d3 + d4;
   float tmp4 = d3 - d4;

   // Even part
   float tmp10 = tmp0 + tmp3;   // phase 2
   float tmp13 = tmp0 - tmp3;
   float tmp11 = tmp1 + tmp2;
   float tmp12 = tmp1 - tmp2;

   d0 = tmp10 + tmp11;       // phase 3
   d4 = tmp10 - tmp11;

   z1 = (tmp12 + tmp13) * 0.707106781f; // c4
   d2 = tmp13 + z1;       // phase 5
   d6 = tmp13 - z1;

   // Odd part
   tmp10 = tmp4 + tmp5;       // phase 2
   tmp11 = tmp5 + tmp6;
   tmp12 = tmp6 + tmp7;

   // The rotator is modified from fig 4-8 to avoid extra
negations.
   z5 = (tmp10 - tmp12) * 0.382683433f; // c6
   z2 = tmp10 * 0.541196100f + z5; // c2-c6
   z4 = tmp12 * 1.306562965f + z5; // c2+c6
   z3 = tmp11 * 0.707106781f; // c4

   z11 = tmp7 + z3;      // phase 5
   z13 = tmp7 - z3;

   *d5p = z13 + z2;         // phase 6
   *d3p = z13 - z2;
   *d1p = z11 + z4;
   *d7p = z11 - z4;
```

```
   *d0p = d0;  *d2p = d2;  *d4p = d4;  *d6p = d6;
}

static void stbiw__jpg_calcBits(int val, unsigned short
bits[2]) {
   int tmp1 = val < 0 ? -val : val;
   val = val < 0 ? val-1 : val;
   bits[1] = 1;
   while(tmp1 >>= 1) {
      ++bits[1];
   }
   bits[0] = val & ((1<<bits[1])-1);
}

static int stbiw__jpg_processDU(stbi__write_context *s, int
*bitBuf, int *bitCnt, float *CDU, float *fdtbl, int DC,
const unsigned short HTDC[256][2], const unsigned short
HTAC[256][2]) {
   const unsigned short EOB[2] = { HTAC[0x00][0],
HTAC[0x00][1] };
   const unsigned short M16zeroes[2] = { HTAC[0xF0][0],
HTAC[0xF0][1] };
   int dataOff, i, diff, end0pos;
   int DU[64];

   // DCT rows
   for(dataOff=0; dataOff<64; dataOff+=8) {
      stbiw__jpg_DCT(&CDU[dataOff], &CDU[dataOff+1],
&CDU[dataOff+2], &CDU[dataOff+3], &CDU[dataOff+4],
&CDU[dataOff+5], &CDU[dataOff+6], &CDU[dataOff+7]);
   }
   // DCT columns
   for(dataOff=0; dataOff<8; ++dataOff) {
      stbiw__jpg_DCT(&CDU[dataOff], &CDU[dataOff+8],
&CDU[dataOff+16], &CDU[dataOff+24], &CDU[dataOff+32],
&CDU[dataOff+40], &CDU[dataOff+48], &CDU[dataOff+56]);
   }
   // Quantize/descale/zigzag the coefficients
   for(i=0; i<64; ++i) {
      float v = CDU[i]*fdtbl[i];
      // DU[stbiw__jpg_ZigZag[i]] = (int)(v < 0 ? ceilf(v -
0.5f) : floorf(v + 0.5f));
      // ceilf() and floorf() are C99, not C89, but I
/think/ they're not needed here anyway?
      DU[stbiw__jpg_ZigZag[i]] = (int)(v < 0 ? v - 0.5f : v
+ 0.5f);
   }

   // Encode DC
   diff = DU[0] - DC;
   if (diff == 0) {
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, HTDC[0]);
   } else {
      unsigned short bits[2];
      stbiw__jpg_calcBits(diff, bits);
```

```
      stbiw__jpg_writeBits(s, bitBuf, bitCnt,
HTDC[bits[1]]);
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, bits);
   }
   // Encode ACs
   end0pos = 63;
   for(; (end0pos>0)&&(DU[end0pos]==0); --end0pos) {
   }
   // end0pos = first element in reverse order !=0
   if(end0pos == 0) {
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, EOB);
      return DU[0];
   }
   for(i = 1; i <= end0pos; ++i) {
      int startpos = i;
      int nrzeroes;
      unsigned short bits[2];
      for (; DU[i]==0 && i<=end0pos; ++i) {
      }
      nrzeroes = i-startpos;
      if ( nrzeroes >= 16 ) {
         int lng = nrzeroes>>4;
         int nrmarker;
         for (nrmarker=1; nrmarker <= lng; ++nrmarker)
            stbiw__jpg_writeBits(s, bitBuf, bitCnt,
M16zeroes);
         nrzeroes &= 15;
      }
      stbiw__jpg_calcBits(DU[i], bits);
      stbiw__jpg_writeBits(s, bitBuf, bitCnt,
HTAC[(nrzeroes<<4)+bits[1]]);
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, bits);
   }
   if(end0pos != 63) {
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, EOB);
   }
   return DU[0];
}

static int stbi_write_jpg_core(stbi__write_context *s, int
width, int height, int comp, const void* data, int quality)
{
   // Constants that don't pollute global namespace
   static const unsigned char std_dc_luminance_nrcodes[] =
{0,0,1,5,1,1,1,1,1,1,0,0,0,0,0,0,0};
   static const unsigned char std_dc_luminance_values[] =
{0,1,2,3,4,5,6,7,8,9,10,11};
   static const unsigned char std_ac_luminance_nrcodes[] =
{0,0,2,1,3,3,2,4,3,5,5,4,4,0,0,1,0x7d};
   static const unsigned char std_ac_luminance_values[] = {

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

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

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

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

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

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

0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xf1,0xf2,0xf3,0xf4,
0xf5,0xf6,0xf7,0xf8,0xf9,0xfa
   };
   static const unsigned char std_dc_chrominance_nrcodes[] =
{0,0,3,1,1,1,1,1,1,1,1,1,0,0,0,0,0};
   static const unsigned char std_dc_chrominance_values[] =
{0,1,2,3,4,5,6,7,8,9,10,11};
   static const unsigned char std_ac_chrominance_nrcodes[] =
{0,0,2,1,2,4,4,3,4,7,5,4,4,0,1,2,0x77};
   static const unsigned char std_ac_chrominance_values[] =
{

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

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

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

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

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

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

0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xf2,0xf3,0xf4,
0xf5,0xf6,0xf7,0xf8,0xf9,0xfa
   };
   // Huffman tables
   static const unsigned short YDC_HT[256][2] = {
{0,2},{2,3},{3,3},{4,3},{5,3},{6,3},{14,4},{30,5},{62,6},{12
6,7},{254,8},{510,9}};
   static const unsigned short UVDC_HT[256][2] = {
{0,2},{1,2},{2,2},{6,3},{14,4},{30,5},{62,6},{126,7},{254,8}
,{510,9},{1022,10},{2046,11}};
   static const unsigned short YAC_HT[256][2] = {

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

,10},{65410,16},{65411,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,
0},

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

{28,5},{249,8},{1015,10},{4084,12},{65417,16},{65418,16},{65
419,16},{65420,16},{65421,16},{65422,16},{0,0},{0,0},{0,0},{
0,0},{0,0},{0,0},

{58,6},{503,9},{4085,12},{65423,16},{65424,16},{65425,16},{6
5426,16},{65427,16},{65428,16},{65429,16},{0,0},{0,0},{0,0},
{0,0},{0,0},{0,0},

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

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

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

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

{504,9},{32704,15},{65462,16},{65463,16},{65464,16},{65465,1
6},{65466,16},{65467,16},{65468,16},{65469,16},{0,0},{0,0},{
0,0},{0,0},{0,0},{0,0},

{505,9},{65470,16},{65471,16},{65472,16},{65473,16},{65474,1
6},{65475,16},{65476,16},{65477,16},{65478,16},{0,0},{0,0},{
0,0},{0,0},{0,0},{0,0},

{506,9},{65479,16},{65480,16},{65481,16},{65482,16},{65483,1
6},{65484,16},{65485,16},{65486,16},{65487,16},{0,0},{0,0},{
0,0},{0,0},{0,0},{0,0},

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

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

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

{65515,16},{65516,16},{65517,16},{65518,16},{65519,16},{6552

```
0,16},{65521,16},{65522,16},{65523,16},{65524,16},{0,0},{0,0
},{0,0},{0,0},{0,0},

{2041,11},{65525,16},{65526,16},{65527,16},{65528,16},{65529
,16},{65530,16},{65531,16},{65532,16},{65533,16},{65534,16},
{0,0},{0,0},{0,0},{0,0},{0,0}
   };
   static const unsigned short UVAC_HT[256][2] = {

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

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

{26,5},{247,8},{1015,10},{4086,12},{32706,15},{65420,16},{65
421,16},{65422,16},{65423,16},{65424,16},{0,0},{0,0},{0,0},{
0,0},{0,0},{0,0},

{27,5},{248,8},{1016,10},{4087,12},{65425,16},{65426,16},{65
427,16},{65428,16},{65429,16},{65430,16},{0,0},{0,0},{0,0},{
0,0},{0,0},{0,0},

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

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

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

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

{249,8},{65463,16},{65464,16},{65465,16},{65466,16},{65467,1
6},{65468,16},{65469,16},{65470,16},{65471,16},{0,0},{0,0},{
0,0},{0,0},{0,0},{0,0},

{503,9},{65472,16},{65473,16},{65474,16},{65475,16},{65476,1
6},{65477,16},{65478,16},{65479,16},{65480,16},{0,0},{0,0},{
0,0},{0,0},{0,0},{0,0},

{504,9},{65481,16},{65482,16},{65483,16},{65484,16},{65485,1
6},{65486,16},{65487,16},{65488,16},{65489,16},{0,0},{0,0},{
0,0},{0,0},{0,0},{0,0},

{505,9},{65490,16},{65491,16},{65492,16},{65493,16},{65494,1
6},{65495,16},{65496,16},{65497,16},{65498,16},{0,0},{0,0},{
0,0},{0,0},{0,0},{0,0},
```

```
{506,9},{65499,16},{65500,16},{65501,16},{65502,16},{65503,1
6},{65504,16},{65505,16},{65506,16},{65507,16},{0,0},{0,0},{
0,0},{0,0},{0,0},{0,0},

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

{16352,14},{65517,16},{65518,16},{65519,16},{65520,16},{6552
1,16},{65522,16},{65523,16},{65524,16},{65525,16},{0,0},{0,0
},{0,0},{0,0},{0,0},

{1018,10},{32707,15},{65526,16},{65527,16},{65528,16},{65529
,16},{65530,16},{65531,16},{65532,16},{65533,16},{65534,16},
{0,0},{0,0},{0,0},{0,0},{0,0}
   };
   static const int YQT[] =
{16,11,10,16,24,40,51,61,12,12,14,19,26,58,60,55,14,13,16,24
,40,57,69,56,14,17,22,29,51,87,80,62,18,22,

37,56,68,109,103,77,24,35,55,64,81,104,113,92,49,64,78,87,10
3,121,120,101,72,92,95,98,112,100,103,99};
   static const int UVQT[] =
{17,18,24,47,99,99,99,99,18,21,26,66,99,99,99,99,24,26,56,99
,99,99,99,99,47,66,99,99,99,99,99,99,

99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
99,99,99,99,99,99,99,99,99,99,99,99,99,99};
   static const float aasf[] = { 1.0f * 2.828427125f,
1.387039845f * 2.828427125f, 1.306562965f * 2.828427125f,
1.175875602f * 2.828427125f,
                                 1.0f * 2.828427125f,
0.785694958f * 2.828427125f, 0.541196100f * 2.828427125f,
0.275899379f * 2.828427125f };

   int row, col, i, k;
   float fdtbl_Y[64], fdtbl_UV[64];
   unsigned char YTable[64], UVTable[64];

   if(!data || !width || !height || comp > 4 || comp < 1) {
      return 0;
   }

   quality = quality ? quality : 90;
   quality = quality < 1 ? 1 : quality > 100 ? 100 :
quality;
   quality = quality < 50 ? 5000 / quality : 200 - quality *
2;

   for(i = 0; i < 64; ++i) {
      int uvti, yti = (YQT[i]*quality+50)/100;
      YTable[stbiw__jpg_ZigZag[i]] = (unsigned char) (yti <
1 ? 1 : yti > 255 ? 255 : yti);
      uvti = (UVQT[i]*quality+50)/100;
```

```
      UVTable[stbiw__jpg_ZigZag[i]] = (unsigned char) (uvti
< 1 ? 1 : uvti > 255 ? 255 : uvti);
   }

   for(row = 0, k = 0; row < 8; ++row) {
      for(col = 0; col < 8; ++col, ++k) {
         fdtbl_Y[k]  = 1 / (YTable [stbiw__jpg_ZigZag[k]] *
aasf[row] * aasf[col]);
         fdtbl_UV[k] = 1 / (UVTable[stbiw__jpg_ZigZag[k]] *
aasf[row] * aasf[col]);
      }
   }

   // Write Headers
   {
      static const unsigned char head0[] = {
0xFF,0xD8,0xFF,0xE0,0,0x10,'J','F','I','F',0,1,1,0,0,1,0,1,0
,0,0xFF,0xDB,0,0x84,0 };
      static const unsigned char head2[] = {
0xFF,0xDA,0,0xC,3,1,0,2,0x11,3,0x11,0,0x3F,0 };
      const unsigned char head1[] = {
0xFF,0xC0,0,0x11,8,(unsigned
char)(height>>8),STBIW_UCHAR(height),(unsigned
char)(width>>8),STBIW_UCHAR(width),

3,1,0x11,0,2,0x11,1,3,0x11,1,0xFF,0xC4,0x01,0xA2,0 };
      s->func(s->context, (void*)head0, sizeof(head0));
      s->func(s->context, (void*)YTable, sizeof(YTable));
      stbiw__putc(s, 1);
      s->func(s->context, UVTable, sizeof(UVTable));
      s->func(s->context, (void*)head1, sizeof(head1));
      s->func(s->context,
(void*)(std_dc_luminance_nrcodes+1),
sizeof(std_dc_luminance_nrcodes)-1);
      s->func(s->context, (void*)std_dc_luminance_values,
sizeof(std_dc_luminance_values));
      stbiw__putc(s, 0x10); // HTYACinfo
      s->func(s->context,
(void*)(std_ac_luminance_nrcodes+1),
sizeof(std_ac_luminance_nrcodes)-1);
      s->func(s->context, (void*)std_ac_luminance_values,
sizeof(std_ac_luminance_values));
      stbiw__putc(s, 1); // HTUDCinfo
      s->func(s->context,
(void*)(std_dc_chrominance_nrcodes+1),
sizeof(std_dc_chrominance_nrcodes)-1);
      s->func(s->context, (void*)std_dc_chrominance_values,
sizeof(std_dc_chrominance_values));
      stbiw__putc(s, 0x11); // HTUACinfo
      s->func(s->context,
(void*)(std_ac_chrominance_nrcodes+1),
sizeof(std_ac_chrominance_nrcodes)-1);
      s->func(s->context, (void*)std_ac_chrominance_values,
sizeof(std_ac_chrominance_values));
      s->func(s->context, (void*)head2, sizeof(head2));
   }
```

```
   // Encode 8x8 macroblocks
   {
      static const unsigned short fillBits[] = {0x7F, 7};
      const unsigned char *imageData = (const unsigned char
*)data;
      int DCY=0, DCU=0, DCV=0;
      int bitBuf=0, bitCnt=0;
      // comp == 2 is grey+alpha (alpha is ignored)
      int ofsG = comp > 2 ? 1 : 0, ofsB = comp > 2 ? 2 : 0;
      int x, y, pos;
      for(y = 0; y < height; y += 8) {
         for(x = 0; x < width; x += 8) {
            float YDU[64], UDU[64], VDU[64];
            for(row = y, pos = 0; row < y+8; ++row) {
               for(col = x; col < x+8; ++col, ++pos) {
                  int p = row*width*comp + col*comp;
                  float r, g, b;
                  if(row >= height) {
                     p -= width*comp*(row+1 - height);
                  }
                  if(col >= width) {
                     p -= comp*(col+1 - width);
                  }

                  r = imageData[p+0];
                  g = imageData[p+ofsG];
                  b = imageData[p+ofsB];

YDU[pos]=+0.29900f*r+0.58700f*g+0.11400f*b-128;
                  UDU[pos]=-0.16874f*r-
0.33126f*g+0.50000f*b;
                  VDU[pos]=+0.50000f*r-0.41869f*g-
0.08131f*b;
               }
            }

            DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt,
YDU, fdtbl_Y, DCY, YDC_HT, YAC_HT);
            DCU = stbiw__jpg_processDU(s, &bitBuf, &bitCnt,
UDU, fdtbl_UV, DCU, UVDC_HT, UVAC_HT);
            DCV = stbiw__jpg_processDU(s, &bitBuf, &bitCnt,
VDU, fdtbl_UV, DCV, UVDC_HT, UVAC_HT);
         }
      }

      // Do the bit alignment of the EOI marker
      stbiw__jpg_writeBits(s, &bitBuf, &bitCnt, fillBits);
   }

   // EOI
   stbiw__putc(s, 0xFF);
   stbiw__putc(s, 0xD9);

   return 1;
}
```

```
STBIWDEF int stbi_write_jpg_to_func(stbi_write_func *func,
void *context, int x, int y, int comp, const void *data, int
quality)
{
   stbi__write_context s;
   stbi__start_write_callbacks(&s, func, context);
   return stbi_write_jpg_core(&s, x, y, comp, (void *) data,
quality);
}


#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_jpg(char const *filename, int x, int
y, int comp, const void *data, int quality)
{
   stbi__write_context s;
   if (stbi__start_write_file(&s,filename)) {
      int r = stbi_write_jpg_core(&s, x, y, comp, data,
quality);
      stbi__end_write_file(&s);
      return r;
   } else
      return 0;
}
#endif

#endif // STB_IMAGE_WRITE_IMPLEMENTATION

/* Revision history
      1.07  (2017-07-24)
             doc fix
      1.06 (2017-07-23)
             writing JPEG (using Jon Olick's code)
      1.05   ???
      1.04 (2017-03-03)
             monochrome BMP expansion
      1.03   ???
      1.02 (2016-04-02)
             avoid allocating large structures on the stack
      1.01 (2016-01-16)
             STBIW_REALLOC_SIZED: support allocators with no
realloc support
             avoid race-condition in crc initialization
             minor compile issues
      1.00 (2015-09-14)
             installable file IO function
      0.99 (2015-09-13)
             warning fixes; TGA rle support
      0.98 (2015-04-08)
             added STBIW_MALLOC, STBIW_ASSERT etc
      0.97 (2015-01-18)
             fixed HDR asserts, rewrote HDR rle logic
      0.96 (2015-01-17)
             add HDR output
             fix monochrome BMP
```

```
      0.95 (2014-08-17)
                add monochrome TGA output
      0.94 (2014-05-31)
               rename private functions to avoid conflicts
with stb_image.h
      0.93 (2014-05-27)
               warning fixes
      0.92 (2010-08-01)
               casts to unsigned char to fix warnings
      0.91 (2010-07-17)
               first public release
      0.90   first internal release
*/

/*
------------------------------------------------------------
------------------
This software is available under 2 licenses -- choose
whichever you prefer.
------------------------------------------------------------
------------------
ALTERNATIVE A - MIT License
Copyright (c) 2017 Sean Barrett
Permission is hereby granted, free of charge, to any person
obtaining a copy of
this software and associated documentation files (the
"Software"), to deal in
the Software without restriction, including without
limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies
of the Software, and to permit persons to whom the Software
is furnished to do
so, subject to the following conditions:
The above copyright notice and this permission notice shall
be included in all
copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE
SOFTWARE.
------------------------------------------------------------
------------------
ALTERNATIVE B - Public Domain (www.unlicense.org)
This is free and unencumbered software released into the
public domain.
```

```
Anyone is free to copy, modify, publish, use, compile, sell,
or distribute this
software, either in source code form or as a compiled
binary, for any purpose,
commercial or non-commercial, and by any means.
In jurisdictions that recognize copyright laws, the author
or authors of this
software dedicate any and all copyright interest in the
software to the public
domain. We make this dedication for the benefit of the
public at large and to
the detriment of our heirs and successors. We intend this
dedication to be an
overt act of relinquishment in perpetuity of all present and
future rights to
this software under copyright law.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
EVENT SHALL THE
AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN
ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF
OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
------------------------------------------------------------
-----------------
*/
```

Makefile

```
# Make sure ocamlbuild can find opam-managed packages: first
run
#
# eval `opam config env`

# Easiest way to build: using ocamlbuild, which in turn uses
ocamlfind

.PHONY : all
all : clean pixl.native stdlib.o

.PHONY : pixl.native
pixl.native :
   ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -
cflags -w,+a-4 \
        pixl.native

# "make clean" removes all generated files

.PHONY : clean
clean :
   ocamlbuild -clean
```

```
   rm -rf testall.log *.diff pixl scanner.ml parser.ml
parser.mli
   rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.s *.ll *.out *.exe
   rm -rf stdlib *.err

# More detailed: build using ocamlc/ocamlopt + ocamlfind to
locate LLVM

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx
pixl.cmx

pixl : $(OBJS)
   ocamlfind ocamlopt -linkpkg -package llvm -package
llvm.analysis $(OBJS) -o pixl

scanner.ml : scanner.mll
   ocamllex scanner.mll

parser.ml parser.mli : parser.mly
   ocamlyacc parser.mly

%.cmo : %.ml
   ocamlc -c $<

%.cmi : %.mli
   ocamlc -c $<

%.cmx : %.ml
   ocamlfind ocamlopt -c -package llvm $<

stdlib : stdlib.c
   cc -o stdlib -DBUILD_TEST stdlib.c

### Generated by "ocamldep *.ml *.mli" after building
scanner.ml and parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
pixl.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo
ast.cmo
pixl.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx
ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo
```

compile.sh

```
#!/bin/sh

./pixl.native < $1.p > $1.ll
```

```
llc $1.ll
gcc -o $1 $1.s stdlib.o
rm $1.ll $1.s
```

compileandrun.sh

```
#!/bin/sh

./compile.sh $1
./$1
rm $1
```