libpng.txt - A description on how to use and modify libpng

libpng version 1.0.6 - March 21, 2000

Updated and distributed by Glenn Randers-Pehrson

<randeg@alum.rpi.edu>

Copyright (c) 1998, 1999, 2000 Glenn Randers-Pehrson

For conditions of distribution and use, see copyright

notice in png.h.

based on:

libpng 1.0 beta 6 version 0.96 May 28, 1997

Updated and distributed by Andreas Dilger

Copyright (c) 1996, 1997 Andreas Dilger

libpng 1.0 beta 2 - version 0.88 January 26, 1996

For conditions of distribution and use, see copyright

notice in png.h. Copyright (c) 1995, 1996 Guy Eric

Schalnat, Group 42, Inc.

Updated/rewritten per request in the libpng FAQ

Copyright (c) 1995, 1996 Frank J. T. Wojcik

December 18, 1995 & January 20, 1996

I. Introduction

This file describes how to use and modify the PNG reference library

(known as libpng) for your own use. There are five sections to this

file: introduction, structures, reading, writing, and modification and

configuration notes for various special platforms. In addition to this

file, example.c is a good starting point for using the library, as

it is heavily commented and should include everything most people

will need. We assume that libpng is already installed; see the

INSTALL file for instructions on how to install libpng.

Libpng was written as a companion to the PNG specification, as a way

of reducing the amount of time and effort it takes to support the PNG

file format in application programs.

The PNG-1.2 specification is available at <http://www.cdrom.com/pub/png>

(will be moving to <http://www.libpng.org>)

and at <ftp://ftp.uu.net/graphics/png/documents/>.

The PNG-1.0 specification is available

as RFC 2083 <ftp://ftp.uu.net/graphics/png/documents/> and as a

W3C Recommendation <http://www.w3.org/TR/REC.png.html>. Some

additional chunks are described in the special-purpose public chunks

documents at <ftp://ftp.uu.net/graphics/png/documents/>.

Other information

about PNG, and the latest version of libpng, can be found at the PNG home

page, <http://www.cdrom.com/pub/png/> (will be moving to

<http://www.libpng.org>)

and at <ftp://ftp.uu.net/graphics/png/>.

Most users will not have to modify the library significantly; advanced

users may want to modify it more. All attempts were made to make it as

complete as possible, while keeping the code easy to understand.

Currently, this library only supports C. Support for other languages

is being considered.

Libpng has been designed to handle multiple sessions at one time,

to be easily modifiable, to be portable to the vast majority of

machines (ANSI, K&R, 16-, 32-, and 64-bit) available, and to be easy

to use. The ultimate goal of libpng is to promote the acceptance of

the PNG file format in whatever way possible. While there is still

work to be done (see the TODO file), libpng should cover the

majority of the needs of its users.

Libpng uses zlib for its compression and decompression of PNG files.

Further information about zlib, and the latest version of zlib, can

be found at the zlib home page, <ftp://ftp.freesoftware.com/pub/infozip/zlib/>.

The zlib compression utility is a general purpose utility that is

useful for more than PNG files, and can be used without libpng.

See the documentation delivered with zlib for more details.

You can usually find the source files for the zlib utility wherever you

find the libpng source files.

Libpng is thread safe, provided the threads are using different

instances of the structures. Each thread should have its own

png\_struct and png\_info instances, and thus its own image.

Libpng does not protect itself against two threads using the

same instance of a structure.

II. Structures

There are two main structures that are important to libpng, png\_struct

and png\_info. The first, png\_struct, is an internal structure that

will not, for the most part, be used by a user except as the first

variable passed to every libpng function call.

The png\_info structure is designed to provide information about the

PNG file. At one time, the fields of png\_info were intended to be

directly accessible to the user. However, this tended to cause problems

with applications using dynamically loaded libraries, and as a result

a set of interface functions for png\_info was developed. The fields

of png\_info are still available for older applications, but it is

suggested that applications use the new interfaces if at all possible.

The png.h header file is an invaluable reference for programming with libpng.

And while I'm on the topic, make sure you include the libpng header file:

#include <png.h>

III. Reading

We'll now walk you through the possible functions to call when reading

in a PNG file sequentially, briefly explaining the syntax and purpose

of each one. See example.c and png.h for more detail. While

progressive reading is covered in the next section, you will still

need some of the functions discussed in this section to read a PNG

file.

Setup

You will want to do the I/O initialization(\*) before you get into libpng,

so if it doesn't work, you don't have much to undo. Of course, you

will also want to insure that you are, in fact, dealing with a PNG

file. Libpng provides a simple check to see if a file is a PNG file.

To use it, pass in the first 1 to 8 bytes of the file, and it will

return true or false (1 or 0) depending on whether the bytes could be

part of a PNG file. Of course, the more bytes you pass in, the

greater the accuracy of the prediction.

If you are intending to keep the file pointer open for use in libpng,

you must ensure you don't read more than 8 bytes from the beginning

of the file, and you also have to make a call to png\_set\_sig\_bytes\_read()

with the number of bytes you read from the beginning. Libpng will

then only check the bytes (if any) that your program didn't read.

(\*): If you are not using the standard I/O functions, you will need

to replace them with custom functions. See the discussion under

Customizing libpng.

FILE \*fp = fopen(file\_name, "rb");

if (!fp)

{

return;

}

fread(header, 1, number, fp);

is\_png = !png\_sig\_cmp(header, 0, number);

if (!is\_png)

{

return;

}

Next, png\_struct and png\_info need to be allocated and initialized. In

order to ensure that the size of these structures is correct even with a

dynamically linked libpng, there are functions to initialize and

allocate the structures. We also pass the library version, optional

pointers to error handling functions, and a pointer to a data struct for

use by the error functions, if necessary (the pointer and functions can

be NULL if the default error handlers are to be used). See the section

on Changes to Libpng below regarding the old initialization functions.

png\_structp png\_ptr = png\_create\_read\_struct

(PNG\_LIBPNG\_VER\_STRING, (png\_voidp)user\_error\_ptr,

user\_error\_fn, user\_warning\_fn);

if (!png\_ptr)

return;

png\_infop info\_ptr = png\_create\_info\_struct(png\_ptr);

if (!info\_ptr)

{

png\_destroy\_read\_struct(&png\_ptr,

(png\_infopp)NULL, (png\_infopp)NULL);

return;

}

png\_infop end\_info = png\_create\_info\_struct(png\_ptr);

if (!end\_info)

{

png\_destroy\_read\_struct(&png\_ptr, &info\_ptr,

(png\_infopp)NULL);

return;

}

If you want to use your own memory allocation routines,

define PNG\_USER\_MEM\_SUPPORTED and use

png\_create\_read\_struct\_2() instead of png\_create\_read\_struct():

png\_structp png\_ptr = png\_create\_read\_struct\_2

(PNG\_LIBPNG\_VER\_STRING, (png\_voidp)user\_error\_ptr,

user\_error\_fn, user\_warning\_fn, (png\_voidp)

user\_mem\_ptr, user\_malloc\_fn, user\_free\_fn);

The error handling routines passed to png\_create\_read\_struct()

and the memory alloc/free routines passed to png\_create\_struct\_2()

are only necessary if you are not using the libpng supplied error

handling and memory alloc/free functions.

When libpng encounters an error, it expects to longjmp back

to your routine. Therefore, you will need to call setjmp and pass

your png\_jmpbuf(png\_ptr). If you read the file from different

routines, you will need to update the jmpbuf field every time you enter

a new routine that will call a png\_\*() function.

See your documentation of setjmp/longjmp for your compiler for more

information on setjmp/longjmp. See the discussion on libpng error

handling in the Customizing Libpng section below for more information

on the libpng error handling. If an error occurs, and libpng longjmp's

back to your setjmp, you will want to call png\_destroy\_read\_struct() to

free any memory.

if (setjmp(png\_jmpbuf(png\_ptr)))

{

png\_destroy\_read\_struct(&png\_ptr, &info\_ptr,

&end\_info);

fclose(fp);

return;

}

If you would rather avoid the complexity of setjmp/longjmp issues,

you can compile libpng with PNG\_SETJMP\_NOT\_SUPPORTED, in which case

errors will result in a call to PNG\_ABORT() which defaults to abort().

Now you need to set up the input code. The default for libpng is to

use the C function fread(). If you use this, you will need to pass a

valid FILE \* in the function png\_init\_io(). Be sure that the file is

opened in binary mode. If you wish to handle reading data in another

way, you need not call the png\_init\_io() function, but you must then

implement the libpng I/O methods discussed in the Customizing Libpng

section below.

png\_init\_io(png\_ptr, fp);

If you had previously opened the file and read any of the signature from

the beginning in order to see if this was a PNG file, you need to let

libpng know that there are some bytes missing from the start of the file.

png\_set\_sig\_bytes(png\_ptr, number);

Setting up callback code

You can set up a callback function to handle any unknown chunks in the

input stream. You must supply the function

read\_chunk\_callback(png\_ptr ptr,

png\_unknown\_chunkp chunk);

{

/\* The unknown chunk structure contains your

chunk data: \*/

png\_byte name[5];

png\_byte \*data;

png\_size\_t size;

/\* Note that libpng has already taken care of the

CRC handling \*/

/\* put your code here. Return one of the following: \*/

return (-n); /\* chunk had an error \*/

return (0); /\* did not recognize \*/

return (n); /\* success \*/

}

(You can give your function another name that you like instead of

"read\_chunk\_callback")

To inform libpng about your function, use

png\_set\_read\_user\_chunk\_fn(png\_ptr, user\_chunk\_ptr,

read\_chunk\_callback);

This names not only the callback function, but also a user pointer that

you can retrieve with

png\_get\_user\_chunk\_ptr(png\_ptr);

At this point, you can set up a callback function that will be

called after each row has been read, which you can use to control

a progress meter or the like. It's demonstrated in pngtest.c.

You must supply a function

void read\_row\_callback(png\_ptr ptr, png\_uint\_32 row, int pass);

{

/\* put your code here \*/

}

(You can give it another name that you like instead of "read\_row\_callback")

To inform libpng about your function, use

png\_set\_read\_status\_fn(png\_ptr, read\_row\_callback);

Unknown-chunk handling

Now you get to set the way the library processes unknown chunks in the

input PNG stream. Both known and unknown chunks will be read. Normal

behavior is that known chunks will be parsed into information in

various info\_ptr members; unknown chunks will be discarded. To change

this, you can call:

png\_set\_keep\_unknown\_chunks(png\_ptr, info\_ptr, keep,

chunk\_list, num\_chunks);

keep - 0: do not keep

1: keep only if safe-to-copy

2: keep even if unsafe-to-copy

chunk\_list - list of chunks affected (a byte string,

five bytes per chunk, NULL or '\0' if

num\_chunks is 0)

num\_chunks - number of chunks affected; if 0, all

unknown chunks are affected

Unknown chunks declared in this way will be saved as raw data onto a

list of png\_unknown\_chunk structures. If a chunk that is normally

known to libpng is named in the list, it will be handled as unknown,

according to the "keep" directive. If a chunk is named in successive

instances of png\_set\_keep\_unknown\_chunks(), the final instance will

take precedence.

The high-level read interface

At this point there are two ways to proceed; through the high-level

read interface, or through a sequence of low-level read operations.

You can use the high-level interface if (a) you are willing to read

the entire image into memory, and (b) the input transformations

you want to do are limited to the following set:

PNG\_TRANSFORM\_IDENTITY No transformation

PNG\_TRANSFORM\_STRIP\_16 Strip 16-bit samples to 8 bits

PNG\_TRANSFORM\_STRIP\_ALPHA Discard the alpha channel

PNG\_TRANSFORM\_PACKING Expand 1, 2 and 4-bit samples to bytes

PNG\_TRANSFORM\_PACKSWAP Change order of packed pixels to LSB first

PNG\_TRANSFORM\_EXPAND Perform set\_expand()

PNG\_TRANSFORM\_INVERT\_MONO Invert monochrome images

PNG\_TRANSFORM\_SHIFT Normalize pixels to the sBIT depth

PNG\_TRANSFORM\_BGR Flip RGB to BGR, RGBA to BGRA

PNG\_TRANSFORM\_SWAP\_ALPHA Flip RGBA to ARGB or GA to AG

PNG\_TRANSFORM\_INVERT\_ALPHA Change alpha from opacity to transparency

PNG\_TRANSFORM\_SWAP\_ENDIAN Byte-swap 16-bit samples

(This excludes setting a background color, doing gamma transformation,

dithering, and setting filler.) If this is the case, simply do this:

png\_read\_png(png\_ptr, info\_ptr, png\_transforms, NULL)

where png\_transforms is an integer containing the logical-or of some set of

transformation flags. This call is equivalent to png\_read\_info(),

followed the set of transformations indicated by the transform mask,

followed by png\_update\_info(), followed by a read of the image bytes

to the info member `rowpointers', followed by png\_read\_end().

(The final parameter of this call is not yet used. Someday it

will point to transformation parameters.)

The low-level read interface

If you are going the low-level route, you are now ready to read all

the file information up to the actual image data. You do this with a

call to png\_read\_info().

png\_read\_info(png\_ptr, info\_ptr);

This will process all chunks up to but not including the image data.

Querying the info structure

Functions are used to get the information from the info\_ptr once it

has been read. Note that these fields may not be completely filled

in until png\_read\_end() has read the chunk data following the image.

png\_get\_IHDR(png\_ptr, info\_ptr, &width, &height,

&bit\_depth, &color\_type, &interlace\_type,

&compression\_type, &filter\_type);

width - holds the width of the image

in pixels (up to 2^31).

height - holds the height of the image

in pixels (up to 2^31).

bit\_depth - holds the bit depth of one of the

image channels. (valid values are

1, 2, 4, 8, 16 and depend also on

the color\_type. See also

significant bits (sBIT) below).

color\_type - describes which color/alpha channels

are present.

PNG\_COLOR\_TYPE\_GRAY

(bit depths 1, 2, 4, 8, 16)

PNG\_COLOR\_TYPE\_GRAY\_ALPHA

(bit depths 8, 16)

PNG\_COLOR\_TYPE\_PALETTE

(bit depths 1, 2, 4, 8)

PNG\_COLOR\_TYPE\_RGB

(bit\_depths 8, 16)

PNG\_COLOR\_TYPE\_RGB\_ALPHA

(bit\_depths 8, 16)

PNG\_COLOR\_MASK\_PALETTE

PNG\_COLOR\_MASK\_COLOR

PNG\_COLOR\_MASK\_ALPHA

filter\_type - (must be PNG\_FILTER\_TYPE\_BASE

for PNG 1.0)

compression\_type - (must be PNG\_COMPRESSION\_TYPE\_BASE

for PNG 1.0)

interlace\_type - (PNG\_INTERLACE\_NONE or

PNG\_INTERLACE\_ADAM7)

Any or all of interlace\_type, compression\_type, of

filter\_type can be

NULL if you are not interested in their values.

channels = png\_get\_channels(png\_ptr, info\_ptr);

channels - number of channels of info for the

color type (valid values are 1 (GRAY,

PALETTE), 2 (GRAY\_ALPHA), 3 (RGB),

4 (RGB\_ALPHA or RGB + filler byte))

rowbytes = png\_get\_rowbytes(png\_ptr, info\_ptr);

rowbytes - number of bytes needed to hold a row

signature = png\_get\_signature(png\_ptr, info\_ptr);

signature - holds the signature read from the

file (if any). The data is kept in

the same offset it would be if the

whole signature were read (i.e. if an

application had already read in 4

bytes of signature before starting

libpng, the remaining 4 bytes would

be in signature[4] through signature[7]

(see png\_set\_sig\_bytes())).

width = png\_get\_image\_width(png\_ptr,

info\_ptr);

height = png\_get\_image\_height(png\_ptr,

info\_ptr);

bit\_depth = png\_get\_bit\_depth(png\_ptr,

info\_ptr);

color\_type = png\_get\_color\_type(png\_ptr,

info\_ptr);

filter\_type = png\_get\_filter\_type(png\_ptr,

info\_ptr);

compression\_type = png\_get\_compression\_type(png\_ptr,

info\_ptr);

interlace\_type = png\_get\_interlace\_type(png\_ptr,

info\_ptr);

These are also important, but their validity depends on whether the chunk

has been read. The png\_get\_valid(png\_ptr, info\_ptr, PNG\_INFO\_<chunk>) and

png\_get\_<chunk>(png\_ptr, info\_ptr, ...) functions return non-zero if the

data has been read, or zero if it is missing. The parameters to the

png\_get\_<chunk> are set directly if they are simple data types, or a pointer

into the info\_ptr is returned for any complex types.

png\_get\_PLTE(png\_ptr, info\_ptr, &palette,

&num\_palette);

palette - the palette for the file

(array of png\_color)

num\_palette - number of entries in the palette

png\_get\_gAMA(png\_ptr, info\_ptr, &gamma);

gamma - the gamma the file is written

at (PNG\_INFO\_gAMA)

png\_get\_sRGB(png\_ptr, info\_ptr, &srgb\_intent);

srgb\_intent - the rendering intent (PNG\_INFO\_sRGB)

The presence of the sRGB chunk

means that the pixel data is in the

sRGB color space. This chunk also

implies specific values of gAMA and

cHRM.

png\_get\_iCCP(png\_ptr, info\_ptr, &name, &compression\_type,

&profile, &proflen);

name - The profile name.

compression - The compression type; always PNG\_COMPRESSION\_TYPE\_BASE

for PNG 1.0. You may give NULL to this argument

to ignore it.

profile - International Color Consortium color profile

data. May contain NULs.

proflen - length of profile data in bytes.

png\_get\_sBIT(png\_ptr, info\_ptr, &sig\_bit);

sig\_bit - the number of significant bits for

(PNG\_INFO\_sBIT) each of the gray,

red, green, and blue channels,

whichever are appropriate for the

given color type (png\_color\_16)

png\_get\_tRNS(png\_ptr, info\_ptr, &trans, &num\_trans,

&trans\_values);

trans - array of transparent entries for

palette (PNG\_INFO\_tRNS)

trans\_values - transparent pixel for non-paletted

images (PNG\_INFO\_tRNS)

num\_trans - number of transparent entries

(PNG\_INFO\_tRNS)

png\_get\_hIST(png\_ptr, info\_ptr, &hist);

(PNG\_INFO\_hIST)

hist - histogram of palette (array of

png\_color\_16)

png\_get\_tIME(png\_ptr, info\_ptr, &mod\_time);

mod\_time - time image was last modified

(PNG\_VALID\_tIME)

png\_get\_bKGD(png\_ptr, info\_ptr, &background);

background - background color (PNG\_VALID\_bKGD)

valid 16-bit red, green and blue

values, regardless of color\_type

num\_comments = png\_get\_text(png\_ptr, info\_ptr,

&text\_ptr, &num\_text);

num\_comments - number of comments

text\_ptr - array of png\_text holding image

comments

text\_ptr[i]->compression - type of compression used

on "text" PNG\_TEXT\_COMPRESSION\_NONE

PNG\_TEXT\_COMPRESSION\_zTXt

PNG\_ITXT\_COMPRESSION\_NONE

PNG\_ITXT\_COMPRESSION\_zTXt

text\_ptr[i]->key - keyword for comment.

text\_ptr[i]->text - text comments for current

keyword.

text\_ptr[i]->text\_length - length of text string,

after decompression, 0 for iTXt

text\_ptr[i]->itxt\_length - length of itxt string,

after decompression, 0 for tEXt/zTXt

text\_ptr[i]->lang - language of comment (NULL for unknown).

text\_ptr[i]->translated\_keyword - keyword in UTF-8 (NULL

for unknown).

num\_text - number of comments (same as num\_comments;

you can put NULL here to avoid the duplication)

num\_spalettes = png\_get\_spalettes(png\_ptr, info\_ptr, &palette\_ptr);

palette\_ptr - array of png\_spalette structures holding contents

of one or more sPLT chunks read.

num\_spalettes - number of sPLT chunks read.

png\_get\_oFFs(png\_ptr, info\_ptr, &offset\_x, &offset\_y,

&unit\_type);

offset\_x - positive offset from the left edge

of the screen

offset\_y - positive offset from the top edge

of the screen

unit\_type - PNG\_OFFSET\_PIXEL, PNG\_OFFSET\_MICROMETER

png\_get\_pHYs(png\_ptr, info\_ptr, &res\_x, &res\_y,

&unit\_type);

res\_x - pixels/unit physical resolution in

x direction

res\_y - pixels/unit physical resolution in

x direction

unit\_type - PNG\_RESOLUTION\_UNKNOWN,

PNG\_RESOLUTION\_METER

png\_get\_sCAL(png\_ptr, info\_ptr, &unit, &width, &height)

unit - physical scale units (a string)

width - width of a pixel in physical scale units

height - height of a pixel in physical scale units

num\_unknown\_chunks = png\_get\_unknown\_chunks(png\_ptr, info\_ptr,

&unknowns)

unknowns - array of png\_unknown\_chunk structures holding

unknown chunks

unknowns[i].name - name of unknown chunk

unknowns[i].data - data of unknown chunk

unknowns[i].size - size of unknown chunk

unknowns[i].location - position of chunk in file

The data from the pHYs chunk can be retrieved in several convenient

forms:

res\_x = png\_get\_x\_pixels\_per\_meter(png\_ptr,

info\_ptr)

res\_y = png\_get\_y\_pixels\_per\_meter(png\_ptr,

info\_ptr)

res\_x\_and\_y = png\_get\_pixels\_per\_meter(png\_ptr,

info\_ptr)

aspect\_ratio = png\_get\_pixel\_aspect\_ratio(png\_ptr,

info\_ptr)

(Each of these returns 0 [signifying "unknown"] if

the data is not present or if res\_x is 0;

res\_x\_and\_y is 0 if res\_x != res\_y)

For more information, see the png\_info definition in png.h and the

PNG specification for chunk contents. Be careful with trusting

rowbytes, as some of the transformations could increase the space

needed to hold a row (expand, filler, gray\_to\_rgb, etc.).

See png\_read\_update\_info(), below.

A quick word about text\_ptr and num\_text. PNG stores comments in

keyword/text pairs, one pair per chunk, with no limit on the number

of text chunks, and a 2^31 byte limit on their size. While there are

suggested keywords, there is no requirement to restrict the use to these

strings. It is strongly suggested that keywords and text be sensible

to humans (that's the point), so don't use abbreviations. Non-printing

symbols are not allowed. See the PNG specification for more details.

There is also no requirement to have text after the keyword.

Keywords should be limited to 79 Latin-1 characters without leading or

trailing spaces, but non-consecutive spaces are allowed within the

keyword. It is possible to have the same keyword any number of times.

The text\_ptr is an array of png\_text structures, each holding a

pointer to a language string, a pointer to a keyword and a pointer to

a text string. Only the text string may be null. The keyword/text

pairs are put into the array in the order that they are received.

However, some or all of the text chunks may be after the image, so, to

make sure you have read all the text chunks, don't mess with these

until after you read the stuff after the image. This will be

mentioned again below in the discussion that goes with png\_read\_end().

Input transformations

After you've read the header information, you can set up the library

to handle any special transformations of the image data. The various

ways to transform the data will be described in the order that they

should occur. This is important, as some of these change the color

type and/or bit depth of the data, and some others only work on

certain color types and bit depths. Even though each transformation

checks to see if it has data that it can do something with, you should

make sure to only enable a transformation if it will be valid for the

data. For example, don't swap red and blue on grayscale data.

The colors used for the background and transparency values should be

supplied in the same format/depth as the current image data. They

are stored in the same format/depth as the image data in a bKGD or tRNS

chunk, so this is what libpng expects for this data. The colors are

transformed to keep in sync with the image data when an application

calls the png\_read\_update\_info() routine (see below).

Data will be decoded into the supplied row buffers packed into bytes

unless the library has been told to transform it into another format.

For example, 4 bit/pixel paletted or grayscale data will be returned

2 pixels/byte with the leftmost pixel in the high-order bits of the

byte, unless png\_set\_packing() is called. 8-bit RGB data will be stored

in RGB RGB RGB format unless png\_set\_filler() is called to insert filler

bytes, either before or after each RGB triplet. 16-bit RGB data will

be returned RRGGBB RRGGBB, with the most significant byte of the color

value first, unless png\_set\_strip\_16() is called to transform it to

regular RGB RGB triplets, or png\_set\_filler() is called to insert

filler bytes, either before or after each RRGGBB triplet. Similarly,

8-bit or 16-bit grayscale data can be modified with png\_set\_filler()

or png\_set\_strip\_16().

The following code transforms grayscale images of less than 8 to 8 bits,

changes paletted images to RGB, and adds a full alpha channel if there is

transparency information in a tRNS chunk. This is most useful on

grayscale images with bit depths of 2 or 4 or if there is a multiple-image

viewing application that wishes to treat all images in the same way.

if (color\_type == PNG\_COLOR\_TYPE\_PALETTE &&

bit\_depth <= 8) png\_set\_palette\_to\_rgb(png\_ptr);

if (color\_type == PNG\_COLOR\_TYPE\_GRAY &&

bit\_depth < 8) png\_set\_gray\_1\_2\_4\_to\_8(png\_ptr);

if (png\_get\_valid(png\_ptr, info\_ptr,

PNG\_INFO\_tRNS)) png\_set\_tRNS\_to\_alpha(png\_ptr);

These three functions are actually aliases for png\_set\_expand(), added

in libpng version 1.0.4, with the function names expanded to improve code

readability. In some future version they may actually do different

things.

PNG can have files with 16 bits per channel. If you only can handle

8 bits per channel, this will strip the pixels down to 8 bit.

if (bit\_depth == 16)

png\_set\_strip\_16(png\_ptr);

The png\_set\_background() function tells libpng to composite images

with alpha or simple transparency against the supplied background

color. If the PNG file contains a bKGD chunk (PNG\_INFO\_bKGD valid),

you may use this color, or supply another color more suitable for

the current display (e.g., the background color from a web page). You

need to tell libpng whether the color is in the gamma space of the

display (PNG\_BACKGROUND\_GAMMA\_SCREEN for colors you supply), the file

(PNG\_BACKGROUND\_GAMMA\_FILE for colors from the bKGD chunk), or one

that is neither of these gammas (PNG\_BACKGROUND\_GAMMA\_UNIQUE - I don't

know why anyone would use this, but it's here).

If, for some reason, you don't need the alpha channel on an image,

and you want to remove it rather than combining it with the background

(but the image author certainly had in mind that you \*would\* combine

it with the background, so that's what you should probably do):

if (color\_type & PNG\_COLOR\_MASK\_ALPHA)

png\_set\_strip\_alpha(png\_ptr);

In PNG files, the alpha channel in an image

is the level of opacity. If you need the alpha channel in an image to

be the level of transparency instead of opacity, you can invert the

alpha channel (or the tRNS chunk data) after it's read, so that 0 is

fully opaque and 255 (in 8-bit or paletted images) or 65535 (in 16-bit

images) is fully transparent, with

png\_set\_invert\_alpha(png\_ptr);

PNG files pack pixels of bit depths 1, 2, and 4 into bytes as small as

they can, resulting in, for example, 8 pixels per byte for 1 bit

files. This code expands to 1 pixel per byte without changing the

values of the pixels:

if (bit\_depth < 8)

png\_set\_packing(png\_ptr);

PNG files have possible bit depths of 1, 2, 4, 8, and 16. All pixels

stored in a PNG image have been "scaled" or "shifted" up to the next

higher possible bit depth (e.g. from 5 bits/sample in the range [0,31] to

8 bits/sample in the range [0, 255]). However, it is also possible to

convert the PNG pixel data back to the original bit depth of the image.

This call reduces the pixels back down to the original bit depth:

png\_color\_16p sig\_bit;

if (png\_get\_sBIT(png\_ptr, info\_ptr, &sig\_bit))

png\_set\_shift(png\_ptr, sig\_bit);

PNG files store 3-color pixels in red, green, blue order. This code

changes the storage of the pixels to blue, green, red:

if (color\_type == PNG\_COLOR\_TYPE\_RGB ||

color\_type == PNG\_COLOR\_TYPE\_RGB\_ALPHA)

png\_set\_bgr(png\_ptr);

PNG files store RGB pixels packed into 3 bytes. This code expands them

into 4 bytes for windowing systems that need them in this format:

if (bit\_depth == 8 && color\_type ==

PNG\_COLOR\_TYPE\_RGB) png\_set\_filler(png\_ptr,

filler, PNG\_FILLER\_BEFORE);

where "filler" is the 8 or 16-bit number to fill with, and the location is

either PNG\_FILLER\_BEFORE or PNG\_FILLER\_AFTER, depending upon whether

you want the filler before the RGB or after. This transformation

does not affect images that already have full alpha channels.

If you are reading an image with an alpha channel, and you need the

data as ARGB instead of the normal PNG format RGBA:

if (color\_type == PNG\_COLOR\_TYPE\_RGB\_ALPHA)

png\_set\_swap\_alpha(png\_ptr);

For some uses, you may want a grayscale image to be represented as

RGB. This code will do that conversion:

if (color\_type == PNG\_COLOR\_TYPE\_GRAY ||

color\_type == PNG\_COLOR\_TYPE\_GRAY\_ALPHA)

png\_set\_gray\_to\_rgb(png\_ptr);

Conversely, you can convert an RGB or RGBA image to grayscale or grayscale

with alpha.

if (color\_type == PNG\_COLOR\_TYPE\_RGB ||

color\_type == PNG\_COLOR\_TYPE\_RGB\_ALPHA)

png\_set\_rgb\_to\_gray\_fixed(png\_ptr, error\_action,

int red\_weight, int green\_weight);

error\_action = 1: silently do the conversion

error\_action = 2: issue a warning if the original

image has any pixel where

red != green or red != blue

error\_action = 3: issue an error and abort the

conversion if the original

image has any pixel where

red != green or red != blue

red\_weight: weight of red component times 100000

green\_weight: weight of green component times 100000

If either weight is negative, default

weights (21268, 71514) are used.

If you have set error\_action = 1 or 2, you can

later check whether the image really was gray, after processing

the image rows, with the png\_get\_rgb\_to\_gray\_status(png\_ptr) function.

It will return a png\_byte that is zero if the image was gray or

1 if there were any non-gray pixels. bKGD and sBIT data

will be silently converted to grayscale, using the green channel

data, regardless of the error\_action setting.

With red\_weight+green\_weight<=100000,

the normalized graylevel is computed:

int rw = red\_weight \* 65536;

int gw = green\_weight \* 65536;

int bw = 65536 - (rw + gw);

gray = (rw\*red + gw\*green + bw\*blue)/65536;

The default values approximate those recommended in the Charles

Poynton's Color FAQ, <http://www.inforamp.net/~poynton/>

Copyright (c) 1998-01-04 Charles Poynton poynton@inforamp.net

Y = 0.212671 \* R + 0.715160 \* G + 0.072169 \* B

Libpng approximates this with

Y = 0.21268 \* R + 0.7151 \* G + 0.07217 \* B

which can be expressed with integers as

Y = (6969 \* R + 23434 \* G + 2365 \* B)/32768

The calculation is done in a linear colorspace, if the image gamma

is known.

If you have a grayscale and you are using png\_set\_expand\_depth() or

png\_set\_expand() to change to

a higher bit-depth, you must either supply the background color as a gray

value at the original file bit-depth (need\_expand = 1) or else supply the

background color as an RGB triplet at the final, expanded bit depth

(need\_expand = 0). Similarly, if you are reading a paletted image, you

must either supply the background color as a palette index (need\_expand = 1)

or as an RGB triplet that may or may not be in the palette (need\_expand = 0).

png\_color\_16 my\_background;

png\_color\_16p image\_background;

if (png\_get\_bKGD(png\_ptr, info\_ptr, &image\_background))

png\_set\_background(png\_ptr, image\_background,

PNG\_BACKGROUND\_GAMMA\_FILE, 1, 1.0);

else

png\_set\_background(png\_ptr, &my\_background,

PNG\_BACKGROUND\_GAMMA\_SCREEN, 0, 1.0);

To properly display PNG images on any kind of system, the application needs

to know what the display gamma is. Ideally, the user will know this, and

the application will allow them to set it. One method of allowing the user

to set the display gamma separately for each system is to check for a

SCREEN\_GAMMA or DISPLAY\_GAMMA environment variable, which will hopefully be

correctly set.

Note that display\_gamma is the overall gamma correction required to produce

pleasing results, which depends on the lighting conditions in the surrounding

environment. In a dim or brightly lit room, no compensation other than

the physical gamma exponent of the monitor is needed, while in a dark room

a slightly smaller exponent is better.

double gamma, screen\_gamma;

if (/\* We have a user-defined screen

gamma value \*/)

{

screen\_gamma = user\_defined\_screen\_gamma;

}

/\* One way that applications can share the same

screen gamma value \*/

else if ((gamma\_str = getenv("SCREEN\_GAMMA"))

!= NULL)

{

screen\_gamma = (double)atof(gamma\_str);

}

/\* If we don't have another value \*/

else

{

screen\_gamma = 2.2; /\* A good guess for a

PC monitor in a bright office or a dim room \*/

screen\_gamma = 2.0; /\* A good guess for a

PC monitor in a dark room \*/

screen\_gamma = 1.7 or 1.0; /\* A good

guess for Mac systems \*/

}

The png\_set\_gamma() function handles gamma transformations of the data.

Pass both the file gamma and the current screen\_gamma. If the file does

not have a gamma value, you can pass one anyway if you have an idea what

it is (usually 0.45455 is a good guess for GIF images on PCs). Note

that file gammas are inverted from screen gammas. See the discussions

on gamma in the PNG specification for an excellent description of what

gamma is, and why all applications should support it. It is strongly

recommended that PNG viewers support gamma correction.

if (png\_get\_gAMA(png\_ptr, info\_ptr, &gamma))

png\_set\_gamma(png\_ptr, screen\_gamma, gamma);

else

png\_set\_gamma(png\_ptr, screen\_gamma, 0.45455);

If you need to reduce an RGB file to a paletted file, or if a paletted

file has more entries then will fit on your screen, png\_set\_dither()

will do that. Note that this is a simple match dither that merely

finds the closest color available. This should work fairly well with

optimized palettes, and fairly badly with linear color cubes. If you

pass a palette that is larger then maximum\_colors, the file will

reduce the number of colors in the palette so it will fit into

maximum\_colors. If there is a histogram, it will use it to make

more intelligent choices when reducing the palette. If there is no

histogram, it may not do as good a job.

if (color\_type & PNG\_COLOR\_MASK\_COLOR)

{

if (png\_get\_valid(png\_ptr, info\_ptr,

PNG\_INFO\_PLTE))

{

png\_color\_16p histogram;

png\_get\_hIST(png\_ptr, info\_ptr,

&histogram);

png\_set\_dither(png\_ptr, palette, num\_palette,

max\_screen\_colors, histogram, 1);

}

else

{

png\_color std\_color\_cube[MAX\_SCREEN\_COLORS] =

{ ... colors ... };

png\_set\_dither(png\_ptr, std\_color\_cube,

MAX\_SCREEN\_COLORS, MAX\_SCREEN\_COLORS,

NULL,0);

}

}

PNG files describe monochrome as black being zero and white being one.

The following code will reverse this (make black be one and white be

zero):

if (bit\_depth == 1 && color\_type == PNG\_COLOR\_GRAY)

png\_set\_invert\_mono(png\_ptr);

PNG files store 16 bit pixels in network byte order (big-endian,

ie. most significant bits first). This code changes the storage to the

other way (little-endian, i.e. least significant bits first, the

way PCs store them):

if (bit\_depth == 16)

png\_set\_swap(png\_ptr);

If you are using packed-pixel images (1, 2, or 4 bits/pixel), and you

need to change the order the pixels are packed into bytes, you can use:

if (bit\_depth < 8)

png\_set\_packswap(png\_ptr);

Finally, you can write your own transformation function if none of

the existing ones meets your needs. This is done by setting a callback

with

png\_set\_read\_user\_transform\_fn(png\_ptr,

read\_transform\_fn);

You must supply the function

void read\_transform\_fn(png\_ptr ptr, row\_info\_ptr

row\_info, png\_bytep data)

See pngtest.c for a working example. Your function will be called

after all of the other transformations have been processed.

You can also set up a pointer to a user structure for use by your

callback function, and you can inform libpng that your transform

function will change the number of channels or bit depth with the

function

png\_set\_user\_transform\_info(png\_ptr, user\_ptr,

user\_depth, user\_channels);

The user's application, not libpng, is responsible for allocating and

freeing any memory required for the user structure.

You can retrieve the pointer via the function

png\_get\_user\_transform\_ptr(). For example:

voidp read\_user\_transform\_ptr =

png\_get\_user\_transform\_ptr(png\_ptr);

The last thing to handle is interlacing; this is covered in detail below,

but you must call the function here if you want libpng to handle expansion

of the interlaced image.

number\_of\_passes = png\_set\_interlace\_handling(png\_ptr);

After setting the transformations, libpng can update your png\_info

structure to reflect any transformations you've requested with this

call. This is most useful to update the info structure's rowbytes

field so you can use it to allocate your image memory. This function

will also update your palette with the correct screen\_gamma and

background if these have been given with the calls above.

png\_read\_update\_info(png\_ptr, info\_ptr);

After you call png\_read\_update\_info(), you can allocate any

memory you need to hold the image. The row data is simply

raw byte data for all forms of images. As the actual allocation

varies among applications, no example will be given. If you

are allocating one large chunk, you will need to build an

array of pointers to each row, as it will be needed for some

of the functions below.

Reading image data

After you've allocated memory, you can read the image data.

The simplest way to do this is in one function call. If you are

allocating enough memory to hold the whole image, you can just

call png\_read\_image() and libpng will read in all the image data

and put it in the memory area supplied. You will need to pass in

an array of pointers to each row.

This function automatically handles interlacing, so you don't need

to call png\_set\_interlace\_handling() or call this function multiple

times, or any of that other stuff necessary with png\_read\_rows().

png\_read\_image(png\_ptr, row\_pointers);

where row\_pointers is:

png\_bytep row\_pointers[height];

You can point to void or char or whatever you use for pixels.

If you don't want to read in the whole image at once, you can

use png\_read\_rows() instead. If there is no interlacing (check

interlace\_type == PNG\_INTERLACE\_NONE), this is simple:

png\_read\_rows(png\_ptr, row\_pointers, NULL,

number\_of\_rows);

where row\_pointers is the same as in the png\_read\_image() call.

If you are doing this just one row at a time, you can do this with

a single row\_pointer instead of an array of row\_pointers:

png\_bytep row\_pointer = row;

png\_read\_row(png\_ptr, row\_pointers, NULL);

If the file is interlaced (info\_ptr->interlace\_type != 0), things get

somewhat harder. The only current (PNG Specification version 1.2)

interlacing type for PNG is (interlace\_type == PNG\_INTERLACE\_ADAM7)

is a somewhat complicated 2D interlace scheme, known as Adam7, that

breaks down an image into seven smaller images of varying size, based

on an 8x8 grid.

libpng can fill out those images or it can give them to you "as is".

If you want them filled out, there are two ways to do that. The one

mentioned in the PNG specification is to expand each pixel to cover

those pixels that have not been read yet (the "rectangle" method).

This results in a blocky image for the first pass, which gradually

smooths out as more pixels are read. The other method is the "sparkle"

method, where pixels are drawn only in their final locations, with the

rest of the image remaining whatever colors they were initialized to

before the start of the read. The first method usually looks better,

but tends to be slower, as there are more pixels to put in the rows.

If you don't want libpng to handle the interlacing details, just call

png\_read\_rows() seven times to read in all seven images. Each of the

images is a valid image by itself, or they can all be combined on an

8x8 grid to form a single image (although if you intend to combine them

you would be far better off using the libpng interlace handling).

The first pass will return an image 1/8 as wide as the entire image

(every 8th column starting in column 0) and 1/8 as high as the original

(every 8th row starting in row 0), the second will be 1/8 as wide

(starting in column 4) and 1/8 as high (also starting in row 0). The

third pass will be 1/4 as wide (every 4th pixel starting in column 0) and

1/8 as high (every 8th row starting in row 4), and the fourth pass will

be 1/4 as wide and 1/4 as high (every 4th column starting in column 2,

and every 4th row starting in row 0). The fifth pass will return an

image 1/2 as wide, and 1/4 as high (starting at column 0 and row 2),

while the sixth pass will be 1/2 as wide and 1/2 as high as the original

(starting in column 1 and row 0). The seventh and final pass will be as

wide as the original, and 1/2 as high, containing all of the odd

numbered scanlines. Phew!

If you want libpng to expand the images, call this before calling

png\_start\_read\_image() or png\_read\_update\_info():

if (interlace\_type == PNG\_INTERLACE\_ADAM7)

number\_of\_passes

= png\_set\_interlace\_handling(png\_ptr);

This will return the number of passes needed. Currently, this

is seven, but may change if another interlace type is added.

This function can be called even if the file is not interlaced,

where it will return one pass.

If you are not going to display the image after each pass, but are

going to wait until the entire image is read in, use the sparkle

effect. This effect is faster and the end result of either method

is exactly the same. If you are planning on displaying the image

after each pass, the "rectangle" effect is generally considered the

better looking one.

If you only want the "sparkle" effect, just call png\_read\_rows() as

normal, with the third parameter NULL. Make sure you make pass over

the image number\_of\_passes times, and you don't change the data in the

rows between calls. You can change the locations of the data, just

not the data. Each pass only writes the pixels appropriate for that

pass, and assumes the data from previous passes is still valid.

png\_read\_rows(png\_ptr, row\_pointers, NULL,

number\_of\_rows);

If you only want the first effect (the rectangles), do the same as

before except pass the row buffer in the third parameter, and leave

the second parameter NULL.

png\_read\_rows(png\_ptr, NULL, row\_pointers,

number\_of\_rows);

Finishing a sequential read

After you are finished reading the image through either the high- or

low-level interfaces, you can finish reading the file. If you are

interested in comments or time, which may be stored either before or

after the image data, you should pass the separate png\_info struct if

you want to keep the comments from before and after the image

separate. If you are not interested, you can pass NULL.

png\_read\_end(png\_ptr, end\_info);

When you are done, you can free all memory allocated by libpng like this:

png\_destroy\_read\_struct(&png\_ptr, &info\_ptr,

&end\_info);

It is also possible to individually free the info\_ptr members that

point to libpng-allocated storage with the following functions:

png\_free\_data(png\_ptr, info\_ptr, mask, n)

mask - identifies data to be freed, a mask

made up by the OR one or more of

PNG\_FREE\_PLTE, PNG\_FREE\_TRNS,

PNG\_FREE\_HIST, PNG\_FREE\_ICCP,

PNG\_FREE\_SPLT, PNG\_FREE\_ROWS,

PNG\_FREE\_PCAL, PNG\_FREE\_SCAL,

PNG\_FREE\_TEXT, PNG\_FREE\_UNKN,

or simply PNG\_FREE\_ALL

n - sequence number of item to be freed

(-1 for all items)

These functions may be safely called when the relevant storage has

already been freed, or has not yet been allocated, and will in that

case do nothing. The "n" parameter is ignored if only one item

of the selected data type, such as PLTE, is allowed. If "n" is not

-1, and multiple items are allowed for the data type identified in

the mask, such as text or splt, only the n'th item is freed.

For a more compact example of reading a PNG image, see the file example.c.

Reading PNG files progressively

The progressive reader is slightly different then the non-progressive

reader. Instead of calling png\_read\_info(), png\_read\_rows(), and

png\_read\_end(), you make one call to png\_process\_data(), which calls

callbacks when it has the info, a row, or the end of the image. You

set up these callbacks with png\_set\_progressive\_read\_fn(). You don't

have to worry about the input/output functions of libpng, as you are

giving the library the data directly in png\_process\_data(). I will

assume that you have read the section on reading PNG files above,

so I will only highlight the differences (although I will show

all of the code).

png\_structp png\_ptr;

png\_infop info\_ptr;

/\* An example code fragment of how you would

initialize the progressive reader in your

application. \*/

int

initialize\_png\_reader()

{

png\_ptr = png\_create\_read\_struct

(PNG\_LIBPNG\_VER\_STRING, (png\_voidp)user\_error\_ptr,

user\_error\_fn, user\_warning\_fn);

if (!png\_ptr)

return -1;

info\_ptr = png\_create\_info\_struct(png\_ptr);

if (!info\_ptr)

{

png\_destroy\_read\_struct(&png\_ptr, (png\_infopp)NULL,

(png\_infopp)NULL);

return -1;

}

if (setjmp(png\_jmpbuf(png\_ptr)))

{

png\_destroy\_read\_struct(&png\_ptr, &info\_ptr,

(png\_infopp)NULL);

return -1;

}

/\* This one's new. You can provide functions

to be called when the header info is valid,

when each row is completed, and when the image

is finished. If you aren't using all functions,

you can specify NULL parameters. Even when all

three functions are NULL, you need to call

png\_set\_progressive\_read\_fn(). You can use

any struct as the user\_ptr (cast to a void pointer

for the function call), and retrieve the pointer

from inside the callbacks using the function

png\_get\_progressive\_ptr(png\_ptr);

which will return a void pointer, which you have

to cast appropriately.

\*/

png\_set\_progressive\_read\_fn(png\_ptr, (void \*)user\_ptr,

info\_callback, row\_callback, end\_callback);

return 0;

}

/\* A code fragment that you call as you receive blocks

of data \*/

int

process\_data(png\_bytep buffer, png\_uint\_32 length)

{

if (setjmp(png\_jmpbuf(png\_ptr)))

{

png\_destroy\_read\_struct(&png\_ptr, &info\_ptr,

(png\_infopp)NULL);

return -1;

}

/\* This one's new also. Simply give it a chunk

of data from the file stream (in order, of

course). On machines with segmented memory

models machines, don't give it any more than

64K. The library seems to run fine with sizes

of 4K. Although you can give it much less if

necessary (I assume you can give it chunks of

1 byte, I haven't tried less then 256 bytes

yet). When this function returns, you may

want to display any rows that were generated

in the row callback if you don't already do

so there.

\*/

png\_process\_data(png\_ptr, info\_ptr, buffer, length);

return 0;

}

/\* This function is called (as set by

png\_set\_progressive\_read\_fn() above) when enough data

has been supplied so all of the header has been

read.

\*/

void

info\_callback(png\_structp png\_ptr, png\_infop info)

{

/\* Do any setup here, including setting any of

the transformations mentioned in the Reading

PNG files section. For now, you \_must\_ call

either png\_start\_read\_image() or

png\_read\_update\_info() after all the

transformations are set (even if you don't set

any). You may start getting rows before

png\_process\_data() returns, so this is your

last chance to prepare for that.

\*/

}

/\* This function is called when each row of image

data is complete \*/

void

row\_callback(png\_structp png\_ptr, png\_bytep new\_row,

png\_uint\_32 row\_num, int pass)

{

/\* If the image is interlaced, and you turned

on the interlace handler, this function will

be called for every row in every pass. Some

of these rows will not be changed from the

previous pass. When the row is not changed,

the new\_row variable will be NULL. The rows

and passes are called in order, so you don't

really need the row\_num and pass, but I'm

supplying them because it may make your life

easier.

For the non-NULL rows of interlaced images,

you must call png\_progressive\_combine\_row()

passing in the row and the old row. You can

call this function for NULL rows (it will just

return) and for non-interlaced images (it just

does the memcpy for you) if it will make the

code easier. Thus, you can just do this for

all cases:

\*/

png\_progressive\_combine\_row(png\_ptr, old\_row,

new\_row);

/\* where old\_row is what was displayed for

previously for the row. Note that the first

pass (pass == 0, really) will completely cover

the old row, so the rows do not have to be

initialized. After the first pass (and only

for interlaced images), you will have to pass

the current row, and the function will combine

the old row and the new row.

\*/

}

void

end\_callback(png\_structp png\_ptr, png\_infop info)

{

/\* This function is called after the whole image

has been read, including any chunks after the

image (up to and including the IEND). You

will usually have the same info chunk as you

had in the header, although some data may have

been added to the comments and time fields.

Most people won't do much here, perhaps setting

a flag that marks the image as finished.

\*/

}

IV. Writing

Much of this is very similar to reading. However, everything of

importance is repeated here, so you won't have to constantly look

back up in the reading section to understand writing.

Setup

You will want to do the I/O initialization before you get into libpng,

so if it doesn't work, you don't have anything to undo. If you are not

using the standard I/O functions, you will need to replace them with

custom writing functions. See the discussion under Customizing libpng.

FILE \*fp = fopen(file\_name, "wb");

if (!fp)

{

return;

}

Next, png\_struct and png\_info need to be allocated and initialized.

As these can be both relatively large, you may not want to store these

on the stack, unless you have stack space to spare. Of course, you

will want to check if they return NULL. If you are also reading,

you won't want to name your read structure and your write structure

both "png\_ptr"; you can call them anything you like, such as

"read\_ptr" and "write\_ptr". Look at pngtest.c, for example.

png\_structp png\_ptr = png\_create\_write\_struct

(PNG\_LIBPNG\_VER\_STRING, (png\_voidp)user\_error\_ptr,

user\_error\_fn, user\_warning\_fn);

if (!png\_ptr)

return;

png\_infop info\_ptr = png\_create\_info\_struct(png\_ptr);

if (!info\_ptr)

{

png\_destroy\_write\_struct(&png\_ptr,

(png\_infopp)NULL);

return;

}

If you want to use your own memory allocation routines,

define PNG\_USER\_MEM\_SUPPORTED and use

png\_create\_write\_struct\_2() instead of png\_create\_read\_struct():

png\_structp png\_ptr = png\_create\_write\_struct\_2

(PNG\_LIBPNG\_VER\_STRING, (png\_voidp)user\_error\_ptr,

user\_error\_fn, user\_warning\_fn, (png\_voidp)

user\_mem\_ptr, user\_malloc\_fn, user\_free\_fn);

After you have these structures, you will need to set up the

error handling. When libpng encounters an error, it expects to

longjmp() back to your routine. Therefore, you will need to call

setjmp() and pass the png\_jmpbuf(png\_ptr). If you

write the file from different routines, you will need to update

the png\_jmpbuf(png\_ptr) every time you enter a new routine that will

call a png\_\*() function. See your documentation of setjmp/longjmp

for your compiler for more information on setjmp/longjmp. See

the discussion on libpng error handling in the Customizing Libpng

section below for more information on the libpng error handling.

if (setjmp(png\_jmpbuf(png\_ptr)))

{

png\_destroy\_write\_struct(&png\_ptr, &info\_ptr);

fclose(fp);

return;

}

...

return;

If you would rather avoid the complexity of setjmp/longjmp issues,

you can compile libpng with PNG\_SETJMP\_NOT\_SUPPORTED, in which case

errors will result in a call to PNG\_ABORT() which defaults to abort().

Now you need to set up the output code. The default for libpng is to

use the C function fwrite(). If you use this, you will need to pass a

valid FILE \* in the function png\_init\_io(). Be sure that the file is

opened in binary mode. Again, if you wish to handle writing data in

another way, see the discussion on libpng I/O handling in the Customizing

Libpng section below.

png\_init\_io(png\_ptr, fp);

Write callbacks

At this point, you can set up a callback function that will be

called after each row has been written, which you can use to control

a progress meter or the like. It's demonstrated in pngtest.c.

You must supply a function

void write\_row\_callback(png\_ptr, png\_uint\_32 row, int pass);

{

/\* put your code here \*/

}

(You can give it another name that you like instead of "write\_row\_callback")

To inform libpng about your function, use

png\_set\_write\_status\_fn(png\_ptr, write\_row\_callback);

You now have the option of modifying how the compression library will

run. The following functions are mainly for testing, but may be useful

in some cases, like if you need to write PNG files extremely fast and

are willing to give up some compression, or if you want to get the

maximum possible compression at the expense of slower writing. If you

have no special needs in this area, let the library do what it wants by

not calling this function at all, as it has been tuned to deliver a good

speed/compression ratio. The second parameter to png\_set\_filter() is

the filter method, for which the only valid value is '0' (as of the

July 1999 PNG specification, version 1.2). The third parameter is a

flag that indicates which filter type(s) are to be tested for each

scanline. See the Compression Library for details on the specific filter

types.

/\* turn on or off filtering, and/or choose

specific filters \*/

png\_set\_filter(png\_ptr, 0,

PNG\_FILTER\_NONE | PNG\_FILTER\_SUB |

PNG\_FILTER\_PAETH);

The png\_set\_compression\_\*() functions interface to the zlib compression

library, and should mostly be ignored unless you really know what you are

doing. The only generally useful call is png\_set\_compression\_level()

which changes how much time zlib spends on trying to compress the image

data. See the Compression Library for details on the compression levels.

/\* set the zlib compression level \*/

png\_set\_compression\_level(png\_ptr,

Z\_BEST\_COMPRESSION);

/\* set other zlib parameters \*/

png\_set\_compression\_mem\_level(png\_ptr, 8);

png\_set\_compression\_strategy(png\_ptr,

Z\_DEFAULT\_STRATEGY);

png\_set\_compression\_window\_bits(png\_ptr, 15);

png\_set\_compression\_method(png\_ptr, 8);

Setting the contents of info for output

You now need to fill in the png\_info structure with all the data you

wish to write before the actual image. Note that the only thing you

are allowed to write after the image is the text chunks and the time

chunk (as of PNG Specification 1.2, anyway). See png\_write\_end() and

the latest PNG specification for more information on that. If you

wish to write them before the image, fill them in now, and flag that

data as being valid. If you want to wait until after the data, don't

fill them until png\_write\_end(). For all the fields in png\_info and

their data types, see png.h. For explanations of what the fields

contain, see the PNG specification.

Some of the more important parts of the png\_info are:

png\_set\_IHDR(png\_ptr, info\_ptr, width, height,

bit\_depth, color\_type, interlace\_type,

compression\_type, filter\_type)

width - holds the width of the image

in pixels (up to 2^31).

height - holds the height of the image

in pixels (up to 2^31).

bit\_depth - holds the bit depth of one of the

image channels.

(valid values are 1, 2, 4, 8, 16

and depend also on the

color\_type. See also significant

bits (sBIT) below).

color\_type - describes which color/alpha

channels are present.

PNG\_COLOR\_TYPE\_GRAY

(bit depths 1, 2, 4, 8, 16)

PNG\_COLOR\_TYPE\_GRAY\_ALPHA

(bit depths 8, 16)

PNG\_COLOR\_TYPE\_PALETTE

(bit depths 1, 2, 4, 8)

PNG\_COLOR\_TYPE\_RGB

(bit\_depths 8, 16)

PNG\_COLOR\_TYPE\_RGB\_ALPHA

(bit\_depths 8, 16)

PNG\_COLOR\_MASK\_PALETTE

PNG\_COLOR\_MASK\_COLOR

PNG\_COLOR\_MASK\_ALPHA

interlace\_type - PNG\_INTERLACE\_NONE or

PNG\_INTERLACE\_ADAM7

compression\_type - (must be

PNG\_COMPRESSION\_TYPE\_DEFAULT)

filter\_type - (must be PNG\_FILTER\_TYPE\_DEFAULT)

png\_set\_PLTE(png\_ptr, info\_ptr, palette,

num\_palette);

palette - the palette for the file

(array of png\_color)

num\_palette - number of entries in the palette

png\_set\_gAMA(png\_ptr, info\_ptr, gamma);

gamma - the gamma the image was created

at (PNG\_INFO\_gAMA)

png\_set\_sRGB(png\_ptr, info\_ptr, srgb\_intent);

srgb\_intent - the rendering intent

(PNG\_INFO\_sRGB) The presence of

the sRGB chunk means that the pixel

data is in the sRGB color space.

This chunk also implies specific

values of gAMA and cHRM. Rendering

intent is the CSS-1 property that

has been defined by the International

Color Consortium

(http://www.color.org).

It can be one of

PNG\_sRGB\_INTENT\_SATURATION,

PNG\_sRGB\_INTENT\_PERCEPTUAL,

PNG\_sRGB\_INTENT\_ABSOLUTE, or

PNG\_sRGB\_INTENT\_RELATIVE.

png\_set\_sRGB\_gAMA\_and\_cHRM(png\_ptr, info\_ptr,

srgb\_intent);

srgb\_intent - the rendering intent

(PNG\_INFO\_sRGB) The presence of the

sRGB chunk means that the pixel

data is in the sRGB color space.

This function also causes gAMA and

cHRM chunks with the specific values

that are consistent with sRGB to be

written.

png\_set\_iCCP(png\_ptr, info\_ptr, name, compression\_type,

profile, proflen);

name - The profile name.

compression - The compression type; always PNG\_COMPRESSION\_TYPE\_BASE

for PNG 1.0. You may give NULL to this argument

to ignore it.

profile - International Color Consortium color profile

data. May contain NULs.

proflen - length of profile data in bytes.

png\_set\_sBIT(png\_ptr, info\_ptr, sig\_bit);

sig\_bit - the number of significant bits for

(PNG\_INFO\_sBIT) each of the gray, red,

green, and blue channels, whichever are

appropriate for the given color type

(png\_color\_16)

png\_set\_tRNS(png\_ptr, info\_ptr, trans, num\_trans,

trans\_values);

trans - array of transparent entries for

palette (PNG\_INFO\_tRNS)

trans\_values - transparent pixel for non-paletted

images (PNG\_INFO\_tRNS)

num\_trans - number of transparent entries

(PNG\_INFO\_tRNS)

png\_set\_hIST(png\_ptr, info\_ptr, hist);

(PNG\_INFO\_hIST)

hist - histogram of palette (array of

png\_color\_16)

png\_set\_tIME(png\_ptr, info\_ptr, mod\_time);

mod\_time - time image was last modified

(PNG\_VALID\_tIME)

png\_set\_bKGD(png\_ptr, info\_ptr, background);

background - background color (PNG\_VALID\_bKGD)

png\_set\_text(png\_ptr, info\_ptr, text\_ptr, num\_text);

text\_ptr - array of png\_text holding image

comments

text\_ptr[i]->compression - type of compression used

on "text" PNG\_TEXT\_COMPRESSION\_NONE

PNG\_TEXT\_COMPRESSION\_zTXt

PNG\_ITXT\_COMPRESSION\_NONE

PNG\_ITXT\_COMPRESSION\_zTXt

text\_ptr[i]->key - keyword for comment.

text\_ptr[i]->text - text comments for current

keyword.

text\_ptr[i]->text\_length - length of text string,

after decompression, 0 for iTXt

text\_ptr[i]->itxt\_length - length of itxt string,

after decompression, 0 for tEXt/zTXt

text\_ptr[i]->lang - language of comment (NULL for unknown).

text\_ptr[i]->translated\_keyword - keyword in UTF-8 (NULL

for unknown).

num\_text - number of comments

png\_set\_spalettes(png\_ptr, info\_ptr, &palette\_ptr, num\_spalettes);

palette\_ptr - array of png\_spalette structures to be added to

the list of palettes in the info structure.

num\_spalettes - number of palette structures to be added.

png\_set\_oFFs(png\_ptr, info\_ptr, offset\_x, offset\_y,

unit\_type);

offset\_x - positive offset from the left

edge of the screen

offset\_y - positive offset from the top

edge of the screen

unit\_type - PNG\_OFFSET\_PIXEL, PNG\_OFFSET\_MICROMETER

png\_set\_pHYs(png\_ptr, info\_ptr, res\_x, res\_y,

unit\_type);

res\_x - pixels/unit physical resolution

in x direction

res\_y - pixels/unit physical resolution

in y direction

unit\_type - PNG\_RESOLUTION\_UNKNOWN,

PNG\_RESOLUTION\_METER

png\_set\_sCAL(png\_ptr, info\_ptr, unit, width, height)

unit - physical scale units (a string)

width - width of a pixel in physical scale units

height - height of a pixel in physical scale units

png\_set\_unknown\_chunks(png\_ptr, info\_ptr, &unknowns, num\_unknowns)

unknowns - array of png\_unknown\_chunk structures holding

unknown chunks

unknowns[i].name - name of unknown chunk

unknowns[i].data - data of unknown chunk

unknowns[i].size - size of unknown chunk

unknowns[i].location - position to write chunk in file

0: do not write chunk

PNG\_HAVE\_IHDR: before PLTE

PNG\_HAVE\_PLTE: before IDAT

PNG\_AFTER\_IDAT: after IDAT

The "location" member is set automatically according to

what part of the output file has already been written.

You can change its value after calling png\_set\_unknown\_chunks()

as demonstrated in pngtest.c.

A quick word about text and num\_text. text is an array of png\_text

structures. num\_text is the number of valid structures in the array.

If you want, you can use max\_text to hold the size of the array, but

libpng ignores it for writing (it does use it for reading). Each

png\_text structure holds a language code, a keyword, a text value, and

a compression type.

The compression types have the same valid numbers as the compression

types of the image data. Currently, the only valid number is zero.

However, you can store text either compressed or uncompressed, unlike

images, which always have to be compressed. So if you don't want the

text compressed, set the compression type to PNG\_TEXT\_COMPRESSION\_NONE.

Because compressed-text chunks don't have a language field, if you

specify compression any language code will not be written out.

Until text gets around 1000 bytes, it is not worth compressing it.

After the text has been written out to the file, the compression type

is set to PNG\_TEXT\_COMPRESSION\_NONE\_WR or PNG\_TEXT\_COMPRESSION\_zTXt\_WR,

so that it isn't written out again at the end (in case you are calling

png\_write\_end() with the same struct.

The keywords that are given in the PNG Specification are:

Title Short (one line) title or

caption for image

Author Name of image's creator

Description Description of image (possibly long)

Copyright Copyright notice

Creation Time Time of original image creation

(usually RFC 1123 format, see below)

Software Software used to create the image

Disclaimer Legal disclaimer

Warning Warning of nature of content

Source Device used to create the image

Comment Miscellaneous comment; conversion

from other image format

The keyword-text pairs work like this. Keywords should be short

simple descriptions of what the comment is about. Some typical

keywords are found in the PNG specification, as is some recommendations

on keywords. You can repeat keywords in a file. You can even write

some text before the image and some after. For example, you may want

to put a description of the image before the image, but leave the

disclaimer until after, so viewers working over modem connections

don't have to wait for the disclaimer to go over the modem before

they start seeing the image. Finally, keywords should be full

words, not abbreviations. Keywords and text are in the ISO 8859-1

(Latin-1) character set (a superset of regular ASCII) and can not

contain NUL characters, and should not contain control or other

unprintable characters. To make the comments widely readable, stick

with basic ASCII, and avoid machine specific character set extensions

like the IBM-PC character set. The keyword must be present, but

you can leave off the text string on non-compressed pairs.

Compressed pairs must have a text string, as only the text string

is compressed anyway, so the compression would be meaningless.

PNG supports modification time via the png\_time structure. Two

conversion routines are proved, png\_convert\_from\_time\_t() for

time\_t and png\_convert\_from\_struct\_tm() for struct tm. The

time\_t routine uses gmtime(). You don't have to use either of

these, but if you wish to fill in the png\_time structure directly,

you should provide the time in universal time (GMT) if possible

instead of your local time. Note that the year number is the full

year (e.g. 1998, rather than 98 - PNG is year 2000 compliant!), and

that months start with 1.

If you want to store the time of the original image creation, you should

use a plain tEXt chunk with the "Creation Time" keyword. This is

necessary because the "creation time" of a PNG image is somewhat vague,

depending on whether you mean the PNG file, the time the image was

created in a non-PNG format, a still photo from which the image was

scanned, or possibly the subject matter itself. In order to facilitate

machine-readable dates, it is recommended that the "Creation Time"

tEXt chunk use RFC 1123 format dates (e.g. "22 May 1997 18:07:10 GMT"),

although this isn't a requirement. Unlike the tIME chunk, the

"Creation Time" tEXt chunk is not expected to be automatically changed

by the software. To facilitate the use of RFC 1123 dates, a function

png\_convert\_to\_rfc1123(png\_timep) is provided to convert from PNG

time to an RFC 1123 format string.

Writing unknown chunks

You can use the png\_set\_unknown\_chunks function to queue up chunks

for writing. You give it a chunk name, raw data, and a size; that's

all there is to it. The chunks will be written by the next following

png\_write\_info\_before\_PLTE, png\_write\_info, or png\_write\_end function.

Any chunks previously read into the info structure's unknown-chunk

list will also be written out in a sequence that satisfies the PNG

specification's ordering rules.

The high-level write interface

At this point there are two ways to proceed; through the high-level

write interface, or through a sequence of low-level write operations.

You can use the high-level interface if your image data is present

on the rowpointers member of the info structure. All defined output

transformations are permitted, enabled by the following masks.

PNG\_TRANSFORM\_IDENTITY No transformation

PNG\_TRANSFORM\_PACKING Pack 1, 2 and 4-bit samples

PNG\_TRANSFORM\_PACKSWAP Change order of packed pixels to LSB first

PNG\_TRANSFORM\_INVERT\_MONO Invert monochrome images

PNG\_TRANSFORM\_SHIFT Normalize pixels to the sBIT depth

PNG\_TRANSFORM\_BGR Flip RGB to BGR, RGBA to BGRA

PNG\_TRANSFORM\_SWAP\_ALPHA Flip RGBA to ARGB or GA to AG

PNG\_TRANSFORM\_INVERT\_ALPHA Change alpha from opacity to transparency

PNG\_TRANSFORM\_SWAP\_ENDIAN Byte-swap 16-bit samples

PNG\_TRANSFORM\_STRIP\_FILLER Strip out filler bytes.

If you have valid image data on the rowpointers member, simply do this:

png\_write\_png(png\_ptr, info\_ptr, png\_transforms, NULL)

where png\_transforms is an integer containing the logical-or of some set of

transformation flags. This call is equivalent to png\_write\_info(),

followed by the set of transformations indicated by the transform

mask, followed by followed by a write of the image bytes from the info

member `rowpointers', followed by png\_write\_end().

(The final parameter of this call is not yet used. Someday it

may point to output transformation parameters.)

The low-level write interface

If you are going the low-level route instead, you are now ready to

write all the file information up to the actual image data. You do

this with a call to png\_write\_info().

png\_write\_info(png\_ptr, info\_ptr);

Note that there is one transformation you may need to do before

png\_write\_info(). In PNG files, the alpha channel in an image is the

level of opacity. If your data is supplied as a level of

transparency, you can invert the alpha channel before you write it, so

that 0 is fully transparent and 255 (in 8-bit or paletted images) or

65535 (in 16-bit images) is fully opaque, with

png\_set\_invert\_alpha(png\_ptr);

This must appear before png\_write\_info() instead of later with the

other transformations because in the case of paletted images the tRNS

chunk data has to be inverted before the tRNS chunk is written. If

your image is not a paletted image, the tRNS data (which in such cases

represents a single color to be rendered as transparent) won't need to

be changed, and you can safely do this transformation after your

png\_write\_info() call.

If you need to write a private chunk that you want to appear before

the PLTE chunk when PLTE is present, you can write the PNG info in

two steps, and insert code to write your own chunk between them:

png\_write\_info\_before\_PLTE(png\_ptr, info\_ptr);

png\_set\_unknown\_chunks(png\_ptr, info\_ptr, ...);

png\_write\_info(png\_ptr, info\_ptr);

After you've written the file information, you can set up the library

to handle any special transformations of the image data. The various

ways to transform the data will be described in the order that they

should occur. This is important, as some of these change the color

type and/or bit depth of the data, and some others only work on

certain color types and bit depths. Even though each transformation

checks to see if it has data that it can do something with, you should

make sure to only enable a transformation if it will be valid for the

data. For example, don't swap red and blue on grayscale data.

PNG files store RGB pixels packed into 3 or 6 bytes. This code tells

the library to trip input data that has 4 or 8 bytes per pixel down

to 3 or 6 bytes (or strip 2 or 4-byte grayscale+filler data to 1 or 2

bytes per pixel).

png\_set\_filler(png\_ptr, 0, PNG\_FILLER\_BEFORE);

where the 0 is unused, and the location is either PNG\_FILLER\_BEFORE or

PNG\_FILLER\_AFTER, depending upon whether the filler byte in the is stored

XRGB or RGBX.

PNG files pack pixels of bit depths 1, 2, and 4 into bytes as small as

they can, resulting in, for example, 8 pixels per byte for 1 bit files.

If the data is supplied at 1 pixel per byte, use this code, which will

correctly pack the pixels into a single byte:

png\_set\_packing(png\_ptr);

PNG files reduce possible bit depths to 1, 2, 4, 8, and 16. If your

data is of another bit depth, you can write an sBIT chunk into the

file so that decoders can get the original data if desired.

/\* Set the true bit depth of the image data \*/

if (color\_type & PNG\_COLOR\_MASK\_COLOR)

{

sig\_bit.red = true\_bit\_depth;

sig\_bit.green = true\_bit\_depth;

sig\_bit.blue = true\_bit\_depth;

}

else

{

sig\_bit.gray = true\_bit\_depth;

}

if (color\_type & PNG\_COLOR\_MASK\_ALPHA)

{

sig\_bit.alpha = true\_bit\_depth;

}

png\_set\_sBIT(png\_ptr, info\_ptr, &sig\_bit);

If the data is stored in the row buffer in a bit depth other than

one supported by PNG (e.g. 3 bit data in the range 0-7 for a 4-bit PNG),

this will scale the values to appear to be the correct bit depth as

is required by PNG.

png\_set\_shift(png\_ptr, &sig\_bit);

PNG files store 16 bit pixels in network byte order (big-endian,

ie. most significant bits first). This code would be used if they are

supplied the other way (little-endian, i.e. least significant bits

first, the way PCs store them):

if (bit\_depth > 8)

png\_set\_swap(png\_ptr);

If you are using packed-pixel images (1, 2, or 4 bits/pixel), and you

need to change the order the pixels are packed into bytes, you can use:

if (bit\_depth < 8)

png\_set\_packswap(png\_ptr);

PNG files store 3 color pixels in red, green, blue order. This code

would be used if they are supplied as blue, green, red:

png\_set\_bgr(png\_ptr);

PNG files describe monochrome as black being zero and white being

one. This code would be used if the pixels are supplied with this reversed

(black being one and white being zero):

png\_set\_invert\_mono(png\_ptr);

Finally, you can write your own transformation function if none of

the existing ones meets your needs. This is done by setting a callback

with

png\_set\_write\_user\_transform\_fn(png\_ptr,

write\_transform\_fn);

You must supply the function

void write\_transform\_fn(png\_ptr ptr, row\_info\_ptr

row\_info, png\_bytep data)

See pngtest.c for a working example. Your function will be called

before any of the other transformations are processed.

You can also set up a pointer to a user structure for use by your

callback function.

png\_set\_user\_transform\_info(png\_ptr, user\_ptr, 0, 0);

The user\_channels and user\_depth parameters of this function are ignored

when writing; you can set them to zero as shown.

You can retrieve the pointer via the function

png\_get\_user\_transform\_ptr(). For example:

voidp write\_user\_transform\_ptr =

png\_get\_user\_transform\_ptr(png\_ptr);

It is possible to have libpng flush any pending output, either manually,

or automatically after a certain number of lines have been written. To

flush the output stream a single time call:

png\_write\_flush(png\_ptr);

and to have libpng flush the output stream periodically after a certain

number of scanlines have been written, call:

png\_set\_flush(png\_ptr, nrows);

Note that the distance between rows is from the last time png\_write\_flush()

was called, or the first row of the image if it has never been called.

So if you write 50 lines, and then png\_set\_flush 25, it will flush the

output on the next scanline, and every 25 lines thereafter, unless

png\_write\_flush() is called before 25 more lines have been written.

If nrows is too small (less than about 10 lines for a 640 pixel wide

RGB image) the image compression may decrease noticeably (although this

may be acceptable for real-time applications). Infrequent flushing will

only degrade the compression performance by a few percent over images

that do not use flushing.

Writing the image data

That's it for the transformations. Now you can write the image data.

The simplest way to do this is in one function call. If you have the

whole image in memory, you can just call png\_write\_image() and libpng

will write the image. You will need to pass in an array of pointers to

each row. This function automatically handles interlacing, so you don't

need to call png\_set\_interlace\_handling() or call this function multiple

times, or any of that other stuff necessary with png\_write\_rows().

png\_write\_image(png\_ptr, row\_pointers);

where row\_pointers is:

png\_byte \*row\_pointers[height];

You can point to void or char or whatever you use for pixels.

If you don't want to write the whole image at once, you can

use png\_write\_rows() instead. If the file is not interlaced,

this is simple:

png\_write\_rows(png\_ptr, row\_pointers,

number\_of\_rows);

row\_pointers is the same as in the png\_write\_image() call.

If you are just writing one row at a time, you can do this with

a single row\_pointer instead of an array of row\_pointers:

png\_bytep row\_pointer = row;

png\_write\_row(png\_ptr, row\_pointer);

When the file is interlaced, things can get a good deal more

complicated. The only currently (as of January 2000 -- PNG Specification

version 1.2, dated July 1999) defined interlacing scheme for PNG files

is the "Adam7" interlace scheme, that breaks down an

image into seven smaller images of varying size. libpng will build

these images for you, or you can do them yourself. If you want to

build them yourself, see the PNG specification for details of which

pixels to write when.

If you don't want libpng to handle the interlacing details, just

use png\_set\_interlace\_handling() and call png\_write\_rows() the

correct number of times to write all seven sub-images.

If you want libpng to build the sub-images, call this before you start

writing any rows:

number\_of\_passes =

png\_set\_interlace\_handling(png\_ptr);

This will return the number of passes needed. Currently, this

is seven, but may change if another interlace type is added.

Then write the complete image number\_of\_passes times.

png\_write\_rows(png\_ptr, row\_pointers,

number\_of\_rows);

As some of these rows are not used, and thus return immediately,

you may want to read about interlacing in the PNG specification,

and only update the rows that are actually used.

Finishing a sequential write

After you are finished writing the image, you should finish writing

the file. If you are interested in writing comments or time, you should

pass an appropriately filled png\_info pointer. If you are not interested,

you can pass NULL.

png\_write\_end(png\_ptr, info\_ptr);

When you are done, you can free all memory used by libpng like this:

png\_destroy\_write\_struct(&png\_ptr, &info\_ptr);

It is also possible to individually free the info\_ptr members that

point to libpng-allocated storage with the following functions:

png\_free\_text(png\_ptr, info\_ptr, num)

num - number of text item to be freed (-1 for all items)

png\_free\_hIST(png\_ptr, info\_ptr)

png\_free\_iCCP(png\_ptr, info\_ptr)

png\_free\_pCAL(png\_ptr, info\_ptr)

png\_free\_sCAL(png\_ptr, info\_ptr)

png\_free\_sPLT(png\_ptr, info\_ptr, num)

num - number of suggested-paletted entry to be freed

(-1 for all suggested palettes)

png\_free\_pixels(png\_ptr, info\_ptr)

png\_free\_unknown\_chunk(png\_ptr, info\_ptr, num)

num - number of unknown chunk entry to be freed

(-1 for all suggested palettes)

These functions may be safely called when the relevant storage has

already been freed, or has not yet been allocated, and will in that

case do nothing.

If you allocated data such as a palette that you passed in to libpng with

png\_set\_\*, you must not free it until just before the call to

png\_destroy\_write\_struct().

For a more compact example of writing a PNG image, see the file example.c.

V. Modifying/Customizing libpng:

There are two issues here. The first is changing how libpng does

standard things like memory allocation, input/output, and error handling.

The second deals with more complicated things like adding new chunks,

adding new transformations, and generally changing how libpng works.

All of the memory allocation, input/output, and error handling in libpng

goes through callbacks that are user settable. The default routines are

in pngmem.c, pngrio.c, pngwio.c, and pngerror.c respectively. To change

these functions, call the appropriate png\_set\_\*\_fn() function.

Memory allocation is done through the functions png\_large\_malloc(),

png\_malloc(), png\_realloc(), png\_large\_free(), and png\_free(). These

currently just call the standard C functions. The large functions must

handle exactly 64K, but they don't have to handle more than that. If

your pointers can't access more then 64K at a time, you will want to set

MAXSEG\_64K in zlib.h. Since it is unlikely that the method of handling

memory allocation on a platform will change between applications, these

functions must be modified in the library at compile time.

Input/Output in libpng is done through png\_read() and png\_write(),

which currently just call fread() and fwrite(). The FILE \* is stored in

png\_struct and is initialized via png\_init\_io(). If you wish to change

the method of I/O, the library supplies callbacks that you can set

through the function png\_set\_read\_fn() and png\_set\_write\_fn() at run

time, instead of calling the png\_init\_io() function.

These functions

also provide a void pointer that can be retrieved via the function

png\_get\_io\_ptr(). For example:

png\_set\_read\_fn(png\_structp read\_ptr,

voidp read\_io\_ptr, png\_rw\_ptr read\_data\_fn)

png\_set\_write\_fn(png\_structp write\_ptr,

voidp write\_io\_ptr, png\_rw\_ptr write\_data\_fn,

png\_flush\_ptr output\_flush\_fn);

voidp read\_io\_ptr = png\_get\_io\_ptr(read\_ptr);

voidp write\_io\_ptr = png\_get\_io\_ptr(write\_ptr);

The replacement I/O functions should have prototypes as follows:

void user\_read\_data(png\_structp png\_ptr,

png\_bytep data, png\_uint\_32 length);

void user\_write\_data(png\_structp png\_ptr,

png\_bytep data, png\_uint\_32 length);

void user\_flush\_data(png\_structp png\_ptr);

Supplying NULL for the read, write, or flush functions sets them back

to using the default C stream functions. It is an error to read from

a write stream, and vice versa.

Error handling in libpng is done through png\_error() and png\_warning().

Errors handled through png\_error() are fatal, meaning that png\_error()

should never return to its caller. Currently, this is handled via

setjmp() and longjmp() (unless you have compiled libpng with

PNG\_SETJMP\_NOT\_SUPPORTED, in which case it is handled via PNG\_ABORT()),

but you could change this to do things like exit() if you should wish.

On non-fatal errors, png\_warning() is called

to print a warning message, and then control returns to the calling code.

By default png\_error() and png\_warning() print a message on stderr via

fprintf() unless the library is compiled with PNG\_NO\_STDIO defined. If

you wish to change the behavior of the error functions, you will need to

set up your own message callbacks. These functions are normally supplied

at the time that the png\_struct is created. It is also possible to change

these functions after png\_create\_\*\_struct() has been called by calling:

png\_set\_error\_fn(png\_structp png\_ptr,

png\_voidp error\_ptr, png\_error\_ptr error\_fn,

png\_error\_ptr warning\_fn);

png\_voidp error\_ptr = png\_get\_error\_ptr(png\_ptr);

If NULL is supplied for either error\_fn or warning\_fn, then the libpng

default function will be used, calling fprintf() and/or longjmp() if a

problem is encountered. The replacement error functions should have

parameters as follows:

void user\_error\_fn(png\_structp png\_ptr,

png\_const\_charp error\_msg);

void user\_warning\_fn(png\_structp png\_ptr,

png\_const\_charp warning\_msg);

The motivation behind using setjmp() and longjmp() is the C++ throw and

catch exception handling methods. This makes the code much easier to write,

as there is no need to check every return code of every function call.

However, there are some uncertainties about the status of local variables

after a longjmp, so the user may want to be careful about doing anything after

setjmp returns non-zero besides returning itself. Consult your compiler

documentation for more details.

Custom chunks

If you need to read or write custom chunks, you may need to get deeper

into the libpng code. The library now has mechanisms for storing

and writing chunks of unknown type; you can even declare callbacks

for custom chunks. Hoewver, this may not be good enough if the

library code itself needs to know about interactions between your

chunk and existing `intrinsic' chunks.

If you need to write a new intrinsic chunk, first read the PNG

specification. Acquire a first level of

understanding of how it works. Pay particular attention to the

sections that describe chunk names, and look at how other chunks were

designed, so you can do things similarly. Second, check out the

sections of libpng that read and write chunks. Try to find a chunk

that is similar to yours and use it as a template. More details can

be found in the comments inside the code. It is best to handle unknown

chunks in a generic method, via callback functions, instead of by

modifying libpng functions.

If you wish to write your own transformation for the data, look through

the part of the code that does the transformations, and check out some of

the simpler ones to get an idea of how they work. Try to find a similar

transformation to the one you want to add and copy off of it. More details

can be found in the comments inside the code itself.

Configuring for 16 bit platforms

You may need to change the png\_large\_malloc() and png\_large\_free()

routines in pngmem.c, as these are required to allocate 64K, although

there is already support for many of the common DOS compilers. Also,

you will want to look into zconf.h to tell zlib (and thus libpng) that

it cannot allocate more then 64K at a time. Even if you can, the memory

won't be accessible. So limit zlib and libpng to 64K by defining MAXSEG\_64K.

Configuring for DOS

For DOS users who only have access to the lower 640K, you will

have to limit zlib's memory usage via a png\_set\_compression\_mem\_level()

call. See zlib.h or zconf.h in the zlib library for more information.

Configuring for Medium Model

Libpng's support for medium model has been tested on most of the popular

compilers. Make sure MAXSEG\_64K gets defined, USE\_FAR\_KEYWORD gets

defined, and FAR gets defined to far in pngconf.h, and you should be

all set. Everything in the library (except for zlib's structure) is

expecting far data. You must use the typedefs with the p or pp on

the end for pointers (or at least look at them and be careful). Make

note that the rows of data are defined as png\_bytepp, which is an

unsigned char far \* far \*.

Configuring for gui/windowing platforms:

You will need to write new error and warning functions that use the GUI

interface, as described previously, and set them to be the error and

warning functions at the time that png\_create\_\*\_struct() is called,

in order to have them available during the structure initialization.

They can be changed later via png\_set\_error\_fn(). On some compilers,

you may also have to change the memory allocators (png\_malloc, etc.).

Configuring for compiler xxx:

All includes for libpng are in pngconf.h. If you need to add/change/delete

an include, this is the place to do it. The includes that are not

needed outside libpng are protected by the PNG\_INTERNAL definition,

which is only defined for those routines inside libpng itself. The

files in libpng proper only include png.h, which includes pngconf.h.

Configuring zlib:

There are special functions to configure the compression. Perhaps the

most useful one changes the compression level, which currently uses

input compression values in the range 0 - 9. The library normally

uses the default compression level (Z\_DEFAULT\_COMPRESSION = 6). Tests

have shown that for a large majority of images, compression values in

the range 3-6 compress nearly as well as higher levels, and do so much

faster. For online applications it may be desirable to have maximum speed

(Z\_BEST\_SPEED = 1). With versions of zlib after v0.99, you can also

specify no compression (Z\_NO\_COMPRESSION = 0), but this would create

files larger than just storing the raw bitmap. You can specify the

compression level by calling:

png\_set\_compression\_level(png\_ptr, level);

Another useful one is to reduce the memory level used by the library.

The memory level defaults to 8, but it can be lowered if you are

short on memory (running DOS, for example, where you only have 640K).

png\_set\_compression\_mem\_level(png\_ptr, level);

The other functions are for configuring zlib. They are not recommended

for normal use and may result in writing an invalid PNG file. See

zlib.h for more information on what these mean.

png\_set\_compression\_strategy(png\_ptr,

strategy);

png\_set\_compression\_window\_bits(png\_ptr,

window\_bits);

png\_set\_compression\_method(png\_ptr, method);

Controlling row filtering

If you want to control whether libpng uses filtering or not, which

filters are used, and how it goes about picking row filters, you

can call one of these functions. The selection and configuration

of row filters can have a significant impact on the size and

encoding speed and a somewhat lesser impact on the decoding speed

of an image. Filtering is enabled by default for RGB and grayscale

images (with and without alpha), but not for paletted images nor

for any images with bit depths less than 8 bits/pixel.

The 'method' parameter sets the main filtering method, which is

currently only '0' in the PNG 1.2 specification. The 'filters'

parameter sets which filter(s), if any, should be used for each

scanline. Possible values are PNG\_ALL\_FILTERS and PNG\_NO\_FILTERS

to turn filtering on and off, respectively.

Individual filter types are PNG\_FILTER\_NONE, PNG\_FILTER\_SUB,

PNG\_FILTER\_UP, PNG\_FILTER\_AVG, PNG\_FILTER\_PAETH, which can be bitwise

ORed together '|' to specify one or more filters to use. These

filters are described in more detail in the PNG specification. If

you intend to change the filter type during the course of writing

the image, you should start with flags set for all of the filters

you intend to use so that libpng can initialize its internal

structures appropriately for all of the filter types.

filters = PNG\_FILTER\_NONE | PNG\_FILTER\_SUB

| PNG\_FILTER\_UP;

png\_set\_filter(png\_ptr, PNG\_FILTER\_TYPE\_BASE,

filters);

It is also possible to influence how libpng chooses from among the

available filters. This is done in two ways - by telling it how

important it is to keep the same filter for successive rows, and

by telling it the relative computational costs of the filters.

double weights[3] = {1.5, 1.3, 1.1},

costs[PNG\_FILTER\_VALUE\_LAST] =

{1.0, 1.3, 1.3, 1.5, 1.7};

png\_set\_filter\_selection(png\_ptr,

PNG\_FILTER\_SELECTION\_WEIGHTED, 3,

weights, costs);

The weights are multiplying factors that indicate to libpng that the

row filter should be the same for successive rows unless another row filter

is that many times better than the previous filter. In the above example,

if the previous 3 filters were SUB, SUB, NONE, the SUB filter could have a

"sum of absolute differences" 1.5 x 1.3 times higher than other filters

and still be chosen, while the NONE filter could have a sum 1.1 times

higher than other filters and still be chosen. Unspecified weights are

taken to be 1.0, and the specified weights should probably be declining

like those above in order to emphasize recent filters over older filters.

The filter costs specify for each filter type a relative decoding cost

to be considered when selecting row filters. This means that filters

with higher costs are less likely to be chosen over filters with lower

costs, unless their "sum of absolute differences" is that much smaller.

The costs do not necessarily reflect the exact computational speeds of

the various filters, since this would unduly influence the final image

size.

Note that the numbers above were invented purely for this example and

are given only to help explain the function usage. Little testing has

been done to find optimum values for either the costs or the weights.

Removing unwanted object code

There are a bunch of #define's in pngconf.h that control what parts of

libpng are compiled. All the defines end in \_SUPPORTED. If you are

never going to use a capability, you can change the #define to #undef

before recompiling libpng and save yourself code and data space, or

you can turn off individual capabilities with defines that begin with

PNG\_NO\_.

You can also turn all of the transforms and ancillary chunk capabilities

off en masse with compiler directives that define

PNG\_NO\_READ[or WRITE]\_TRANSFORMS, or PNG\_NO\_READ[or WRITE]\_ANCILLARY\_CHUNKS,

or all four,

along with directives to turn on any of the capabilities that you do

want. The PNG\_NO\_READ[or WRITE]\_TRANSFORMS directives disable

the extra transformations but still leave the library fully capable of reading

and writing PNG files with all known public chunks

Use of the PNG\_NO\_READ[or WRITE]\_ANCILLARY\_CHUNKS directive

produces a library that is incapable of reading or writing ancillary chunks.

If you are not using the progressive reading capability, you can

turn that off with PNG\_NO\_PROGRESSIVE\_READ (don't confuse

this with the INTERLACING capability, which you'll still have).

All the reading and writing specific code are in separate files, so the

linker should only grab the files it needs. However, if you want to

make sure, or if you are building a stand alone library, all the

reading files start with pngr and all the writing files start with

pngw. The files that don't match either (like png.c, pngtrans.c, etc.)

are used for both reading and writing, and always need to be included.

The progressive reader is in pngpread.c

If you are creating or distributing a dynamically linked library (a .so

or DLL file), you should not remove or disable any parts of the library,

as this will cause applications linked with different versions of the

library to fail if they call functions not available in your library.

The size of the library itself should not be an issue, because only

those sections that are actually used will be loaded into memory.

Requesting debug printout

The macro definition PNG\_DEBUG can be used to request debugging

printout. Set it to an integer value in the range 0 to 3. Higher

numbers result in increasing amounts of debugging information. The

information is printed to the "stderr" file, unless another file

name is specified in the PNG\_DEBUG\_FILE macro definition.

When PNG\_DEBUG > 0, the following functions (macros) become available:

png\_debug(level, message)

png\_debug1(level, message, p1)

png\_debug2(level, message, p1, p2)

in which "level" is compared to PNG\_DEBUG to decide whether to print

the message, "message" is the formatted string to be printed,

and p1 and p2 are parameters that are to be embedded in the string

according to printf-style formatting directives. For example,

png\_debug1(2, "foo=%d\n", foo);

is expanded to

if(PNG\_DEBUG > 2)

fprintf(PNG\_DEBUG\_FILE, "foo=%d\n", foo);

When PNG\_DEBUG is defined but is zero, the macros aren't defined, but you

can still use PNG\_DEBUG to control your own debugging:

#ifdef PNG\_DEBUG

fprintf(stderr, ...

#endif

When PNG\_DEBUG = 1, the macros are defined, but only png\_debug statements

having level = 0 will be printed. There aren't any such statements in

this version of libpng, but if you insert some they will be printed.

VI. Changes to Libpng from version 0.88

It should be noted that versions of libpng later than 0.96 are not

distributed by the original libpng author, Guy Schalnat, nor by

Andreas Dilger, who had taken over from Guy during 1996 and 1997, and

distributed versions 0.89 through 0.96, but rather by another member

of the original PNG Group, Glenn Randers-Pehrson. Guy and Andreas are

still alive and well, but they have moved on to other things.

The old libpng functions png\_read\_init(), png\_write\_init(),

png\_info\_init(), png\_read\_destroy(), and png\_write\_destory() have been

moved to PNG\_INTERNAL in version 0.95 to discourage their use. The

preferred method of creating and initializing the libpng structures is

via the png\_create\_read\_struct(), png\_create\_write\_struct(), and

png\_create\_info\_struct() because they isolate the size of the structures

from the application, allow version error checking, and also allow the

use of custom error handling routines during the initialization, which

the old functions do not. The functions png\_read\_destroy() and

png\_write\_destroy() do not actually free the memory that libpng

allocated for these structs, but just reset the data structures, so they

can be used instead of png\_destroy\_read\_struct() and

png\_destroy\_write\_struct() if you feel there is too much system overhead

allocating and freeing the png\_struct for each image read.

Setting the error callbacks via png\_set\_message\_fn() before

png\_read\_init() as was suggested in libpng-0.88 is no longer supported

because this caused applications that do not use custom error functions

to fail if the png\_ptr was not initialized to zero. It is still possible

to set the error callbacks AFTER png\_read\_init(), or to change them with

png\_set\_error\_fn(), which is essentially the same function, but with a

new name to force compilation errors with applications that try to use

the old method.

VII. Y2K Compliance in libpng

March 21, 2000

Since the PNG Development group is an ad-hoc body, we can't make

an official declaration.

This is your unofficial assurance that libpng from version 0.71 and

upward through 1.0.6 are Y2K compliant. It is my belief that earlier

versions were also Y2K compliant.

Libpng only has three year fields. One is a 2-byte unsigned integer that

will hold years up to 65535. The other two hold the date in text

format, and will hold years up to 9999.

The integer is

"png\_uint\_16 year" in png\_time\_struct.

The strings are

"png\_charp time\_buffer" in png\_struct and

"near\_time\_buffer", which is a local character string in png.c.

There are seven time-related functions:

png\_convert\_to\_rfc\_1123() in png.c

(formerly png\_convert\_to\_rfc\_1152() in error)

png\_convert\_from\_struct\_tm() in pngwrite.c, called in pngwrite.c

png\_convert\_from\_time\_t() in pngwrite.c

png\_get\_tIME() in pngget.c

png\_handle\_tIME() in pngrutil.c, called in pngread.c

png\_set\_tIME() in pngset.c

png\_write\_tIME() in pngwutil.c, called in pngwrite.c

All appear to handle dates properly in a Y2K environment. The

png\_convert\_from\_time\_t() function calls gmtime() to convert from system

clock time, which returns (year - 1900), which we properly convert to

the full 4-digit year. There is a possibility that applications using

libpng are not passing 4-digit years into the png\_convert\_to\_rfc\_1123()

function, or that they are incorrectly passing only a 2-digit year

instead of "year - 1900" into the png\_convert\_from\_struct\_tm() function,

but this is not under our control. The libpng documentation has always

stated that it works with 4-digit years, and the APIs have been

documented as such.

The tIME chunk itself is also Y2K compliant. It uses a 2-byte unsigned

integer to hold the year, and can hold years as large as 65535.

zlib, upon which libpng depends, is also Y2K compliant. It contains

no date-related code.

Glenn Randers-Pehrson

libpng maintainer

PNG Development Group