A description on how to use and modify libpng

libpng version 1.6.37, April 2019

source: libpng-manual.txt
in distribution v1.6.37, 2019-04-14

edited for pdf layout, released on 2021-02-13
François Esquirol

Based on:

TABLE OF CONTENTS

I. Introduction

This file describes how to use and modify the PNG reference library (known as libpng) for your own use. 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 configure and install libpng.

For examples of libpng usage, see the files "`example.c`", "`pngtest.c`", and the files in the "`contrib`" directory, all of which are included in the libpng distribution.

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 specification (second edition), November 2003, is available as a W3C Recommendation and as an ISO Standard (ISO/IEC 15948:2004 (E)) at <https://www.w3.org/TR/2003/REC-PNG-20031110/>.
The W3C and ISO documents have identical technical content.

The PNG-1.2 specification is available at <https://png-mng.sourceforge.io/pub/png/spec/1.2/>. It is technically equivalent to the PNG specification (second edition) but has some additional material.

The PNG-1.0 specification is available as RFC 2083 at <https://png-mng.sourceforge.io/pub/png/spec/1.0/> and as a W3C Recommendation at <https://www.w3.org/TR/REC-png-961001>.

Some additional chunks are described in the special-purpose public chunks documents at <http://www.libpng.org/pub/png/spec/register/>

Other information about PNG, and the latest version of libpng, can be found at the PNG home page, <http://www.libpng.org/pub/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, <https://zlib.net/>. 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`. Both are internal structures that are no longer exposed in the libpng interface (as of libpng 1.5.0).

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` (the `png_get_*()` and `png_set_*()` functions) was developed, and direct access to the `png_info` fields was deprecated..

The `png_struct` structure is the object used by the library to decode a single image. As of 1.5.0 this structure is also not exposed.

Almost all libpng APIs require a pointer to a `png_struct` as the first argument. Many (in particular the `png_set` and `png_get` APIs) also require a pointer to `png_info` as the second argument. Some application visible macros defined in `png.h` designed for basic data access (reading and writing integers in the PNG format) don't take a `png_info` pointer, but it's almost always safe to assume that a (`png_struct*`) has to be passed to call an API function.

You can have more than one `png_info` structure associated with an image, as illustrated in `pngtest.c`, one for information valid prior to the **`IDAT`** chunks and another (called "`end_info`" below) for things after them.

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>
```

and also (as of libpng-1.5.0) the zlib header file, if you need it:

```
#include <zlib.h>
```

1. Types

The `png.h` header file defines a number of integral types used by the APIs. Most of these are fairly obvious; for example types corresponding to integers of particular sizes and types for passing color values.

One exception is how non-integral numbers are handled. For application convenience most APIs that take such numbers have C (`double`) arguments; however, internally PNG, and libpng, use 32 bit signed integers and encode the value by multiplying by 100,000. As of libpng 1.5.0 a convenience macro `PNG_FP_1` is defined in `png.h` along with a type (`png_fixed_point`) which is simply (`png_int_32`).

All APIs that take (double) arguments also have a matching API that takes the corresponding fixed point integer arguments. The fixed point API has the same name as the floating point one with "_fixed" appended. The actual range of values permitted in the APIs is frequently less than the full range of (png_fixed_point) (-21474 to +21474). When APIs require a non-negative argument the type is recorded as png_uint_32 above. Consult the header file and the text below for more information.

Special care must be take with **sCAL** chunk handling because the chunk itself uses non-integral values encoded as strings containing decimal floating point numbers. See the comments in the header file.

2. Configuration

The main header file function declarations are frequently protected by C preprocessing directives of the form:

```
    #ifdef PNG_feature_SUPPORTED
    declare-function
    #endif
    ...
    #ifdef PNG_feature_SUPPORTED
    use-function
    #endif
```

The library can be built without support for these APIs, although a standard build will have all implemented APIs. Application programs should check the feature macros before using an API for maximum portability. From libpng 1.5.0 the feature macros set during the build of libpng are recorded in the header file "pnglibconf.h" and this file is always included by png.h.

If you don't need to change the library configuration from the default, skip to the next section ("Reading").

Notice that some of the makefiles in the 'scripts' directory and (in 1.5.0) all of the build project files in the 'projects' directory simply copy scripts/pnglibconf.h.prebuilt to pnglibconf.h. This means that these build systems do not permit easy auto-configuration of the library - they only support the default configuration.

The easiest way to make minor changes to the libpng configuration when auto-configuration is supported is to add definitions to the command line using (typically) CPPFLAGS. For example:

```
CPPFLAGS=-DPNG_NO_FLOATING_ARITHMETIC
```

will change the internal libpng math implementation for *gamma correction* and other arithmetic calculations to fixed point, avoiding the need for fast floating point support. The result can be seen in the generated pnglibconf.h - make sure it contains the changed feature macro setting.

If you need to make more extensive configuration changes - more than one or two feature macro settings - you can either add -DPNG_USER_CONFIG to the build command line and put a list of feature macro settings in pngusr.h or you can set DFA_XTRA (a makefile variable) to a file containing the same information in the form of 'option' settings.

A. Changing *pnglibconf.h*

A variety of methods exist to build libpng. Not all of these support reconfiguration of pnglibconf.h. To reconfigure pnglibconf.h it must either be rebuilt from scripts/pnglibconf.dfa using awk or it must be edited by hand.

Hand editing is achieved by copying scripts/pnglibconf.h.prebuilt to pnglibconf.h and changing the lines defining the supported features, paying very close attention to the 'option' information in scripts/pnglibconf.dfa that describes those features and their requirements. This is easy to get wrong.

B. Configuration using DFA_XTRA

Rebuilding from pnglibconf.dfa is easy if a functioning 'awk', or a later variant such as 'nawk' or 'gawk', is available. The configure build will automatically find an appropriate awk and build pnglibconf.h. The scripts/pnglibconf.mak file contains a set of make rules for doing the same thing if configure is not used, and many of the makefiles in the scripts directory use this approach.

When rebuilding simply write a new file containing changed options and set DFA_XTRA to the name of this file. This causes the build to append the new file to the end of scripts/pnglibconf.dfa. The pngusr.dfa file should contain lines of the following forms:

```
everything = off
```

This turns all *optional features off*. Include it at the start of pngusr.dfa to make it easier to build a minimal configuration. You will need to turn at least some features on afterward to enable either reading or writing code, or both.

```
option feature on
option feature off
```

Enable or disable a single feature. This will automatically enable other features required by a feature that is turned on or disable other features that require a feature which is turned off. Conflicting settings will cause an *error message* to be emitted by awk.

setting feature default value

Changes the default value of setting 'feature' to 'value'. There are a small number of settings listed at the top of pnglibconf.h, they are documented in the source code. Most of these values have performance implications for the library but most of them have no visible effect on the API. Some can also be overridden from the API.

This method of building a customized pnglibconf.h is illustrated in contrib/pngminim/*. See the "$(PNGCONF):" target in the makefile and pngusr.dfa in these directories.

C. Configuration using *PNG_USER_CONFIG*

If -DPNG_USER_CONFIG is added to the CPPFLAGS when pnglibconf.h is built, the file pngusr.h will automatically be included before the options in scripts/pnglibconf.dfa are processed. Your pngusr.h file should contain only macro definitions turning features on or off or setting settings.

Apart from the global setting "everything = off" all the options listed above can be set using macros in pngusr.h:

```
#define PNG_feature_SUPPORTED
```

is equivalent to:

```
option feature on
```

```
#define PNG_NO_feature
```

is equivalent to:

```
option feature off
```

```
#define PNG_feature value
```

is equivalent to:

```
setting feature default value
```

Notice that in both cases, pngusr.dfa and pngusr.h, the contents of the pngusr file you supply override the contents of scripts/pnglibconf.dfa

If confusing or incomprehensible behavior results it is possible to examine the intermediate file pnglibconf.dfn to find the full set of dependency information for each setting and option. Simply locate the feature in the file and read the C comments that precede it.

This method is also illustrated in the contrib/pngminim/* makefiles and pngusr.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.

1. 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 to the function png_sig_cmp(), and it will return 0 (false) if the bytes match the corresponding

bytes of the PNG signature, or *nonzero* (`true`) otherwise. 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()` 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 ERROR;
    }

    if (fread(header, 1, number, fp) != number)
    {
       return ERROR;
    }

    is_png = !png_sig_cmp(header, 0, number);
    if (!is_png)
    {
       return NOT_PNG;
    }
```

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. The structure allocation functions quietly return `NULL` if they fail to create the structure, so your application should check for that.

```
    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 ERROR;

    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 ERROR;
    }
```

If you want to use your own memory allocation routines, use a libpng that was built with `PNG_USER_MEM_SUPPORTED` defined, 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 `longjmp` buffer 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 ERROR;
    }
```

Pass (`png_infopp`) `NULL` instead of `&end_info` if you didn't create an `end_info` structure.

If you would rather avoid the complexity of `setjmp`/`longjmp` issues, you can compile libpng with `PNG_NO_SETJMP`, in which case *errors* will result in a call to `PNG_ABORT()` which defaults to `abort()`.

You can `#define PNG_ABORT()` to a function that does something more useful than `abort()`, as long as your *function does not return*.

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);
```

You can change the zlib compression buffer size to be used while reading compressed data with

```
    png_set_compression_buffer_size(png_ptr, buffer_size);
```

where the default size is 8192 bytes. Note that the buffer size is changed immediately and the buffer is reallocated immediately, instead of setting a flag to be acted upon later.

If you want *CRC errors* to be handled in a different manner than the default, use

```
    png_set_crc_action(png_ptr, crit_action, ancil_action);
```

The values for `png_set_crc_action()` say how libpng is to handle *CRC errors* in ancillary and critical chunks, and whether to use the data contained therein. Starting with libpng-1.6.26, this also governs how an *ADLER32 error* is handled while reading the **`IDAT`** chunk. Note that it is impossible to "*discard*" data in a critical chunk.

```
Choices for (int) crit_action are
   PNG_CRC_DEFAULT      0  error/quit
   PNG_CRC_ERROR_QUIT   1  error/quit
   PNG_CRC_WARN_USE     3  warn/use data
   PNG_CRC_QUIET_USE    4  quiet/use data
   PNG_CRC_NO_CHANGE    5  use the current value
```

```
Choices for (int) ancil_action are
   PNG_CRC_DEFAULT      0  error/quit
   PNG_CRC_ERROR_QUIT   1  error/quit
   PNG_CRC_WARN_DISCARD 2  warn/discard data
   PNG_CRC_WARN_USE     3  warn/use data
   PNG_CRC_QUIET_USE    4  quiet/use data
   PNG_CRC_NO_CHANGE    5  use the current value
```

When the setting for crit_action is `PNG_CRC_QUIET_USE`, the *CRC* and *ADLER32* checksums are not only ignored, but they are not evaluated.

2. 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_structp png_ptr,
         png_unknown_chunkp chunk);
    {
       /* The unknown chunk structure contains your
          chunk data, along with similar data for any other
          unknown chunks: */

           png_byte name[5];
           png_byte *data;
           size_t size;

       /* Note that libpng has already taken care of
          the CRC handling */

       /* put your code here.  Search for your chunk in the
          unknown chunk structure, process it, and 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);
```

If you call the png_set_read_user_chunk_fn() function, then all unknown chunks which the *callback* does not handle will be saved when read. You can cause them to be discarded by returning '1' ("handled") instead of '0'. This behavior will change in libpng 1.7 and the default handling set by the png_set_keep_unknown_chunks() function, described below, will be used when the *callback* returns 0. If you want the existing behavior you should set the global default to PNG_HANDLE_CHUNK_IF_SAFE now; this is compatible with all current versions of libpng and with 1.7. Libpng 1.6 issues a *warning* if you keep the default, or PNG_HANDLE_CHUNK_NEVER, and the callback returns 0.

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_structp png_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);
```

When this function is called the row has already been completely processed and the 'row' and 'pass' refer to the next row to be handled. For the non-interlaced case the row that was just handled is simply one less than the passed in row number, and pass will always be 0. For the interlaced case the same applies unless the row value is 0, in which case the row just handled was the last one from one of the preceding passes. Because interlacing may skip a pass you cannot be sure that the preceding pass is just 'pass-1'; if you really need to know what the last pass is record (row,pass) from the *callback* and use the last recorded value each time.

As with the user transform you can find the output row using the PNG_ROW_FROM_PASS_ROW macro.

3. 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 while unknown chunks will be discarded. This behavior can be wasteful if your application will never use some known chunk types. To change this, you can call:

```
    png_set_keep_unknown_chunks(png_ptr, keep,
        chunk_list, num_chunks);

    keep       - 0: default unknown chunk handling
                 1: ignore; do not keep
                 2: keep only if safe-to-copy
                 3: keep even if unsafe-to-copy

               You can use these definitions:
                 PNG_HANDLE_CHUNK_AS_DEFAULT   0
                 PNG_HANDLE_CHUNK_NEVER        1
                 PNG_HANDLE_CHUNK_IF_SAFE      2
                 PNG_HANDLE_CHUNK_ALWAYS       3

    chunk_list - list of chunks affected (a byte string,
                 five bytes per chunk, NULL or '\0' if
                 num_chunks is positive; ignored if
                 numchunks <= 0).

    num_chunks - number of chunks affected; if 0, all
                 unknown chunks are affected.  If positive,
                 only the chunks in the list are affected,
                 and if negative all unknown chunks and
                 all known chunks except for the IHDR,
                 PLTE, tRNS, IDAT, and IEND 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 **IHDR** and **IEND** chunks should not be named in chunk_list; if they are, libpng will process them normally anyway. If you know that your application will never make use of some particular chunks, use PNG_HANDLE_CHUNK_NEVER (or 1) as demonstrated below.

Here is an example of the usage of png_set_keep_unknown_chunks(), where the private "**vpAg**" chunk will later be processed by a user chunk *callback function*:

```
    png_byte vpAg[5]={118, 112,  65, 103, (png_byte) '\0'};

    #if defined(PNG_UNKNOWN_CHUNKS_SUPPORTED)
      png_byte unused_chunks[]=
      {
        104,  73,  83,  84, (png_byte) '\0',   /* hIST */
        105,  84,  88, 116, (png_byte) '\0',   /* iTXt */
        112,  67,  65,  76, (png_byte) '\0',   /* pCAL */
```

```
        115,  67,  65,  76, (png_byte) '\0',   /* sCAL */
        115,  80,  76,  84, (png_byte) '\0',   /* sPLT */
        116,  73,  77,  69, (png_byte) '\0',   /* tIME */
      };
    #endif

    ...

    #if defined(PNG_UNKNOWN_CHUNKS_SUPPORTED)
      /* ignore all unknown chunks
       * (use global setting "2" for libpng16 and earlier):
       */
      png_set_keep_unknown_chunks(read_ptr, 2, NULL, 0);

      /* except for vpAg: */
      png_set_keep_unknown_chunks(read_ptr, 2, vpAg, 1);

      /* also ignore unused known chunks: */
      png_set_keep_unknown_chunks(read_ptr, 1, unused_chunks,
         (int)(sizeof unused_chunks)/5);
    #endif
```

4. User limits

The PNG specification allows the width and height of an image to be as large as `2^31-1` (`0x7fffffff`), or about 2.147 billion rows and columns. For safety, libpng imposes a default limit of 1 million rows and columns. Larger images will be *rejected* immediately with a `png_error()` call. If you wish to change these limits, you can use

```
png_set_user_limits(png_ptr, width_max, height_max);
```

to set your own limits (libpng may *reject* some very wide images anyway because of potential buffer overflow conditions).

You should put this statement after you create the PNG structure and before calling `png_read_info()`, `png_read_png()`, or `png_process_data()`.

When writing a PNG datastream, put this statement before calling `png_write_info()` or `png_write_png()`.

If you need to retrieve the limits that are being applied, use

```
width_max = png_get_user_width_max(png_ptr);
height_max = png_get_user_height_max(png_ptr);
```

The PNG specification sets no limit on the number of ancillary chunks allowed in a PNG datastream. By default, libpng imposes a limit of a total of 1000 **`sPLT`**, **`tEXt`**, **`iTXt`**, **`zTXt`**, and unknown chunks to be stored. If you have set up both `info_ptr` and `end_info_ptr`, the limit applies separately to each. You can change the limit on the total number of such chunks that will be stored, with

```
png_set_chunk_cache_max(png_ptr, user_chunk_cache_max);
```

where 0x7fffffffL means unlimited. You can retrieve this limit with

```
chunk_cache_max = png_get_chunk_cache_max(png_ptr);
```

Libpng imposes a limit of 8 Megabytes (8,000,000 bytes) on the amount of memory that any chunk other than **IDAT** can occupy, originally or when decompressed (prior to libpng-1.6.32 the limit was only applied to compressed chunks after decompression). You can change this limit with

```
png_set_chunk_malloc_max(png_ptr, user_chunk_malloc_max);
```

and you can retrieve the limit with

```
chunk_malloc_max = png_get_chunk_malloc_max(png_ptr);
```

Any chunks that would cause either of these limits to be exceeded will be ignored.

5. Information about your system

If you intend to display the PNG or to incorporate it in other image data you need to tell libpng information about your display or drawing surface so that libpng can convert the values in the image to match the display.

From libpng-1.5.4 this information can be set before reading the PNG file header. In earlier versions png_set_gamma() existed but behaved incorrectly if called before the PNG file header had been read and png_set_alpha_mode() did not exist.

If you need to support versions prior to libpng-1.5.4 test the version number as illustrated below using "PNG_LIBPNG_VER >= 10504" and follow the procedures described in the appropriate manual page.

You give libpng the encoding expected by your system expressed as a 'gamma' value. You can also specify a default encoding for the PNG file in case the required information is missing from the file. By default libpng assumes that the PNG data matches your system, to keep this default call:

```
png_set_gamma(png_ptr, screen_gamma, output_gamma);
```

or you can use the fixed point equivalent:

```
png_set_gamma_fixed(png_ptr, PNG_FP_1*screen_gamma,
   PNG_FP_1*output_gamma);
```

If you don't know the gamma for your system it is probably 2.2 - a good approximation to the IEC standard for display systems (sRGB). If images are too contrasty or washed out you got the value wrong - check your system documentation!

Many systems permit the system gamma to be changed via a lookup table in the display driver, a few systems, including older Macs, change the response by default. As of 1.5.4 three special values are available to handle common situations:

PNG_DEFAULT_sRGB:	Indicates that the system conforms to the IEC 61966-2-1 standard. This matches almost all systems.
PNG_GAMMA_MAC_18:	Indicates that the system is an older (pre Mac OS 10.6) Apple Macintosh system with the default settings.
PNG_GAMMA_LINEAR:	Just the fixed point value for 1.0 - indicates that the system expects data with no gamma encoding.

You would use the linear (unencoded) value if you need to process the pixel values further because this avoids the need to decode and re-encode each component value whenever arithmetic is performed. A lot of graphics software uses linear values for this reason, often with higher precision component values to preserve overall accuracy.

The output_gamma value expresses how to decode the output values, not how they are encoded. The values used correspond to the normal numbers used to describe the overall gamma of a computer display system; for example 2.2 for an sRGB conformant system. The values are scaled by 100000 in the _fixed version of the API (so 220000 for sRGB.)

The inverse of the value is always used to provide a default for the PNG file encoding if it has no gAMA chunk and if png_set_gamma() has not been called to override the PNG gamma information.

When the ALPHA_OPTIMIZED mode is selected the output gamma is used to encode opaque pixels however pixels with lower alpha values are not encoded, regardless of the output gamma setting.

When the standard Porter Duff handling is requested with mode 1 the output encoding is set to be linear and the output_gamma value is only relevant as a default for input data that has no gamma information. The linear output encoding will be overridden if png_set_gamma() is called - the results may be highly unexpected!

The following numbers are derived from the sRGB standard and the research behind it. sRGB is defined to be approximated by a PNG gAMA chunk value of 0.45455 (1/2.2) for PNG. The value implicitly includes any viewing correction required to take account of any differences in the color environment of the original scene and the intended display environment; the value expresses how to **decode** the image for display, not how the original data was **encoded**.

sRGB provides a peg for the PNG standard by defining a viewing environment. sRGB itself, and earlier TV standards, actually use a more complex transform (a linear portion then a gamma 2.4 power law) than PNG can express. (PNG is limited to simple power laws.) By saying that an image for direct display on an sRGB conformant system should be stored with a gAMA chunk value of 45455 (11.3.3.2 and 11.3.3.5 of the ISO PNG specification) the PNG specification makes it possible to derive values for other display systems and environments.

The Mac value is deduced from the sRGB based on an assumption that the actual extra viewing correction used in early Mac display systems was implemented as a power 1.45 lookup table.

Any system where a programmable lookup table is used or where the behavior of the final display device characteristics can be changed requires system specific code to obtain the current characteristic. However this can be difficult and most PNG gamma correction only requires an approximate value.

By default, if png_set_alpha_mode() is not called, libpng assumes that all values are unencoded, linear, values and that the output device also has a linear characteristic. This is only very rarely correct - it is invariably better to call png_set_alpha_mode() with PNG_DEFAULT_sRGB than rely on the default if you don't know what the right answer is!

The special value PNG_GAMMA_MAC_18 indicates an older Mac system (pre Mac OS 10.6) which used a correction table to implement a somewhat lower gamma on an otherwise sRGB system.

Both these values are reserved (not simple gamma values) in order to allow more precise correction internally in the future.

NOTE: the values can be passed to either the fixed or floating point APIs, but the floating point API will also accept floating point values.

The second thing you may need to tell libpng about is how your system handles alpha channel information. Some, but not all, PNG files contain an alpha channel. To display these files correctly you need to compose the data onto a suitable background, as described in the PNG specification.

Libpng only supports composing onto a single color (using png_set_background; see below). Otherwise you must do the composition yourself and, in this case, you may need to call png_set_alpha_mode:

```
   #if PNG_LIBPNG_VER >= 10504
      png_set_alpha_mode(png_ptr, mode, screen_gamma);
   #else
      png_set_gamma(png_ptr, screen_gamma, 1.0/screen_gamma);
   #endif
```

The screen_gamma value is the same as the argument to png_set_gamma; however, how it affects the output depends on the mode. png_set_alpha_mode() sets the file gamma default to 1/screen_gamma, so normally you don't need to call png_set_gamma. If you need different defaults call png_set_gamma() before png_set_alpha_mode() - if you call it after it will override the settings made by png_set_alpha_mode().

The mode is as follows:

PNG_ALPHA_PNG: The data is encoded according to the PNG specification. Red, green and blue, or gray, components are gamma encoded color values and are not premultiplied by the alpha value. The alpha value is a linear measure of the contribution of the pixel to the corresponding final output pixel.

You should normally use this format if you intend to perform color correction on the color values; most, maybe all, color correction software has no handling for the alpha channel and, anyway, the math to handle pre-multiplied component values is unnecessarily complex.

Before you do any arithmetic on the component values you need to remove the gamma encoding and multiply out the alpha channel. See the PNG specification for more detail. It is important to note that

when an image with an alpha channel is scaled, linear encoded, pre-multiplied component values must be used!

The remaining modes assume you don't need to do any further color correction or that if you do, your color correction software knows all about alpha (it probably doesn't!). They 'associate' the alpha with the color information by storing color channel values that have been scaled by the alpha. The advantage is that the color channels can be resampled (the image can be scaled) in this form. The disadvantage is that normal practice is to store linear, not (gamma) encoded, values and this requires 16-bit channels for still images rather than the 8-bit channels that are just about sufficient if gamma encoding is used. In addition all non-transparent pixel values, including completely opaque ones, must be gamma encoded to produce the final image. These are the '`STANDARD`', '`ASSOCIATED`' or '`PREMULTIPLIED`' modes described below (the latter being the two common names for associated alpha color channels). Note that PNG files always contain non-associated color channels; `png_set_alpha_mode()` with one of the modes causes the decoder to convert the pixels to an associated form before returning them to your application.

Since it is not necessary to perform arithmetic on opaque color values so long as they are not to be resampled and are in the final color space it is possible to optimize the handling of alpha by storing the opaque pixels in the PNG format (adjusted for the output color space) while storing partially opaque pixels in the standard, linear, format. The accuracy required for standard alpha composition is relatively low, because the pixels are isolated, therefore typically the accuracy loss in storing 8-bit linear values is acceptable. (This is not true if the alpha channel is used to simulate transparency over large areas - use 16 bits or the PNG mode in this case!) This is the '`OPTIMIZED`' mode. For this mode a pixel is treated as opaque only if the alpha value is equal to the maximum value.

`PNG_ALPHA_STANDARD`: The data libpng produces is encoded in the standard way assumed by most correctly written graphics software. The gamma encoding will be removed by libpng and the linear component values will be pre-multiplied by the alpha channel.

With this format the final image must be re-encoded to match the display gamma before the image is displayed. If your system doesn't do that, yet still seems to perform arithmetic on the pixels without decoding them, it is broken - check out the modes below.

With `PNG_ALPHA_STANDARD` libpng always produces linear component values, whatever `screen_gamma` you supply. The `screen_gamma` value is, however, used as a default for the file gamma if the PNG file has no gamma information.

If you call `png_set_gamma()` after `png_set_alpha_mode()` you will override the linear encoding. Instead the pre-multiplied pixel values will be gamma encoded but the alpha channel will still be linear. This may actually match the requirements of some broken software, but it is unlikely.

While linear 8-bit data is often used it has insufficient precision for any image with a reasonable dynamic range. To avoid problems, and if your software supports it, use `png_set_expand_16()` to force all components to 16 bits.

`PNG_ALPHA_OPTIMIZED`: This mode is the same as `PNG_ALPHA_STANDARD` except that completely opaque pixels are gamma encoded according to the screen_gamma value. Pixels with alpha less than `1.0` will still have linear components.

Use this format if you have control over your compositing software and so don't do other arithmetic (such as scaling) on the data you get from libpng. Your compositing software can simply copy opaque pixels to the output but still has linear values for the non-opaque pixels.

In normal compositing, where the alpha channel encodes partial pixel coverage (as opposed to broad area translucency), the inaccuracies of the 8-bit representation of non-opaque pixels are irrelevant.

You can also try this format if your software is broken; it might look better.

PNG_ALPHA_BROKEN: This is PNG_ALPHA_STANDARD; however, all component values, including the alpha channel are gamma encoded. This is broken because, in practice, no implementation that uses this choice correctly undoes the encoding before handling alpha composition. Use this choice only if other *serious errors* in the software or hardware you use mandate it. In most cases of broken software or hardware the bug in the final display manifests as a subtle halo around composited parts of the image. You may not even perceive this as a halo; the composited part of the image may simply appear separate from the background, as though it had been cut out of paper and pasted on afterward.

If you don't have to deal with bugs in software or hardware, or if you can fix them, there are three recommended ways of using png_set_alpha_mode():

```
   png_set_alpha_mode(png_ptr, PNG_ALPHA_PNG,
       screen_gamma);
```

You can do color correction on the result (libpng does not currently support color correction internally). When you handle the alpha channel you need to undo the gamma encoding and multiply out the alpha.

```
   png_set_alpha_mode(png_ptr, PNG_ALPHA_STANDARD,
       screen_gamma);
   png_set_expand_16(png_ptr);
```

If you are using the high level interface, don't call png_set_expand_16(); instead pass PNG_TRANSFORM_EXPAND_16 to the interface.

With this mode you can't do color correction, but you can do arithmetic, including composition and scaling, on the data without further processing.

```
   png_set_alpha_mode(png_ptr, PNG_ALPHA_OPTIMIZED,
       screen_gamma);
```

You can avoid the expansion to 16-bit components with this mode, but you lose the ability to scale the image or perform other linear arithmetic. All you can do is compose the result onto a matching output. Since this mode is libpng-specific you also need to write your own composition software.

The following are examples of calls to png_set_alpha_mode to achieve the required overall gamma correction and, where necessary, alpha premultiplication.

```
    png_set_alpha_mode(pp, PNG_ALPHA_PNG, PNG_DEFAULT_sRGB);
```

Choices for the alpha_mode are

```
    PNG_ALPHA_PNG           0 /* according to the PNG standard */
    PNG_ALPHA_STANDARD      1 /* according to Porter/Duff */
```

```
    PNG_ALPHA_ASSOCIATED    1 /* as above; this is the normal practice */
    PNG_ALPHA_PREMULTIPLIED 1 /* as above */
    PNG_ALPHA_OPTIMIZED     2 /* 'PNG' for opaque pixels, else 'STANDARD' */
    PNG_ALPHA_BROKEN        3 /* the alpha channel is gamma encoded */
```

PNG_ALPHA_PNG is the default libpng handling of the alpha channel. It is not pre-multiplied into the color components. In addition the call states that the output is for a sRGB system and causes all PNG files without **gAMA** chunks to be assumed to be encoded using sRGB.

```
    png_set_alpha_mode(pp, PNG_ALPHA_PNG, PNG_GAMMA_MAC);
```

In this case the output is assumed to be something like an sRGB conformant display preceded by a power-law lookup table of power 1.45. This is how early Mac systems behaved.

```
    png_set_alpha_mode(pp, PNG_ALPHA_STANDARD, PNG_GAMMA_LINEAR);
```

This is the classic *Jim Blinn* approach and will work in academic environments where everything is done by the book. It has the shortcoming of assuming that input PNG data with no gamma information is linear – this is unlikely to be correct unless the PNG files were generated locally. Most of the time the output precision will be so low as to show significant banding in dark areas of the image.

```
    png_set_expand_16(pp);
    png_set_alpha_mode(pp, PNG_ALPHA_STANDARD, PNG_DEFAULT_sRGB);
```

This is a somewhat more realistic *Jim Blinn* inspired approach. PNG files are assumed to have the sRGB encoding if not marked with a gamma value and the output is always 16 bits per component. This permits accurate scaling and processing of the data. If you know that your input PNG files were generated locally you might need to replace PNG_DEFAULT_sRGB with the correct value for your system.

```
    png_set_alpha_mode(pp, PNG_ALPHA_OPTIMIZED, PNG_DEFAULT_sRGB);
```

If you just need to composite the PNG image onto an existing background and if you control the code that does this you can use the optimization setting. In this case you just copy completely opaque pixels to the output. For pixels that are not completely transparent (you just skip those) you do the composition math using png_composite or png_composite_16 below then encode the resultant 8-bit or 16-bit values to match the output encoding.

6. Other cases

If neither the PNG nor the standard linear encoding work for you because of the software or hardware you use then you have a big problem. The PNG case will probably result in halos around the image. The linear encoding will probably result in a washed out, too bright, image (it's actually too contrasty.) Try the ALPHA_OPTIMIZED mode above - this will probably substantially reduce the halos. Alternatively try:

```
    png_set_alpha_mode(pp, PNG_ALPHA_BROKEN, PNG_DEFAULT_sRGB);
```

This option will also reduce the halos, but there will be slight dark halos round the opaque parts of the image where the background is light. In the OPTIMIZED mode the halos will be light halos where the background is dark. Take your pick - the halos are unavoidable unless you can get your hardware/software fixed! (The OPTIMIZED approach is slightly faster.)

When the default gamma of PNG files doesn't match the output gamma. If you have PNG files with no gamma information png_set_alpha_mode allows you to provide a default gamma, but it also sets the output gamma to the matching value. If you know your PNG files have a gamma that doesn't match the output you can take advantage of the fact that png_set_alpha_mode always sets the output gamma but only sets the PNG default if it is not already set:

```
    png_set_alpha_mode(pp, PNG_ALPHA_PNG, PNG_DEFAULT_sRGB);
    png_set_alpha_mode(pp, PNG_ALPHA_PNG, PNG_GAMMA_MAC);
```

The first call sets both the default and the output gamma values, the second call overrides the output gamma without changing the default. This is easier than achieving the same effect with png_set_gamma. You must use PNG_ALPHA_PNG for the first call - internal checking in png_set_alpha will fire if more than one call to png_set_alpha_mode and png_set_background is made in the same read operation, however multiple calls with PNG_ALPHA_PNG are ignored.

If you don't need, or can't handle, the alpha channel you can call png_set_background() to remove it by compositing against a fixed color. Don't call png_set_strip_alpha() to do this - it will leave spurious pixel values in transparent parts of this image.

```
   png_set_background(png_ptr, &background_color,
       PNG_BACKGROUND_GAMMA_SCREEN, 0, 1);
```

The background_color is an RGB or grayscale value according to the data format libpng will produce for you. Because you don't yet know the format of the PNG file, if you call png_set_background at this point you must arrange for the format produced by libpng to always have 8-bit or 16-bit components and then store the color as an 8-bit or 16-bit color as appropriate. The color contains separate gray and RGB component values, so you can let libpng produce gray or RGB output according to the input format, but low bit depth grayscale images must always be converted to at least 8-bit format. (Even though *low bit depth grayscale* images *can't have an alpha channel* they *can have a transparent color*!)

You set the transforms you need later, either as flags to the high level interface or libpng API calls for the low level interface. For reference the settings and API calls required are:

```
8-bit values:
   PNG_TRANSFORM_SCALE_16 | PNG_EXPAND
   png_set_expand(png_ptr); png_set_scale_16(png_ptr);

   If you must get exactly the same inaccurate results
   produced by default in versions prior to libpng-1.5.4,
   use PNG_TRANSFORM_STRIP_16 and png_set_strip_16(png_ptr)
   instead.
```

```
16-bit values:
   PNG_TRANSFORM_EXPAND_16
   png_set_expand_16(png_ptr);
```

In either case palette image data will be expanded to RGB. If you just want color data you can add PNG_TRANSFORM_GRAY_TO_RGB or png_set_gray_to_rgb(png_ptr) to the list.

Calling png_set_background before the PNG file header is read will not work prior to libpng-1.5.4. Because the failure may result in unexpected *warnings* or *errors* it is therefore much safer to call png_set_background after the head has been read. Unfortunately this means that prior to libpng-1.5.4 it cannot be used with the high level interface.

7. 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_SCALE_16      Strip 16-bit samples to
                                8-bit accurately
    PNG_TRANSFORM_STRIP_16      Chop 16-bit samples to
                                8-bit less accurately
    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
    PNG_TRANSFORM_GRAY_TO_RGB   Expand grayscale samples
                                to RGB (or GA to RGBA)
    PNG_TRANSFORM_EXPAND_16     Expand samples to 16 bits
```

(This excludes setting a background color, doing gamma transformation, quantizing, 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 bitwise 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, then png_read_image(), and finally png_read_end().

(The final parameter of this call is not yet used. Someday it might point to transformation parameters required by some future input transform.)

You must use png_transforms and not call any png_set_transform() functions when you use png_read_png().

After you have called `png_read_png()`, you can retrieve the image data with

```
   row_pointers = png_get_rows(png_ptr, info_ptr);
```

where `row_pointers` is an array of pointers to the pixel data for each row:

```
   png_bytep row_pointers[height];
```

If you know your *image size* and *pixel size* ahead of time, you can allocate `row_pointers` prior to calling `png_read_png()` with

```
   if (height > PNG_UINT_32_MAX/(sizeof (png_byte)))
      png_error (png_ptr,
          "Image is too tall to process in memory");

   if (width > PNG_UINT_32_MAX/pixel_size)
      png_error (png_ptr,
          "Image is too wide to process in memory");

   row_pointers = png_malloc(png_ptr,
       height*(sizeof (png_bytep)));

   for (int i=0; i<height, i++)
      row_pointers[i]=NULL;  /* security precaution */

   for (int i=0; i<height, i++)
      row_pointers[i]=png_malloc(png_ptr,
          width*pixel_size);

   png_set_rows(png_ptr, info_ptr, &row_pointers);
```

Alternatively you could allocate your image in one big block and define `row_pointers[i]` to point into the proper places in your block, but first be sure that your platform is able to allocate such a large buffer:

```
   /* Guard against integer overflow */
   if (height > PNG_SIZE_MAX/(width*pixel_size)) {
        png_error(png_ptr,"image_data buffer would be too large");
   }

   png_bytep buffer=png_malloc(png_ptr,height*width*pixel_size);

   for (int i=0; i<height, i++)
      row_pointers[i]=buffer+i*width*pixel_size;

   png_set_rows(png_ptr, info_ptr, &row_pointers);
```

If you use `png_set_rows()`, the application is responsible for freeing `row_pointers` (and `row_pointers[i]`, if they were separately allocated).

If you don't allocate `row_pointers` ahead of time, `png_read_png()` will do it, and it'll be free'ed by libpng when you call `png_destroy_*()`.

8. 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.

This also copies some of the data from the PNG file into the *decode structure* for use in later transformations. Important information copied in is:

1) The PNG file gamma from the **gAMA** chunk. This overwrites the default value provided by an earlier call to `png_set_gamma` or `png_set_alpha_mode`.

2) Prior to libpng-1.5.4 the background color from a **bKGd** chunk. This damages the information provided by an earlier call to `png_set_background` resulting in unexpected behavior. Libpng-1.5.4 no longer does this.

3) The number of significant bits in each component value. Libpng uses this to optimize gamma handling by reducing the internal lookup table sizes.

4) The transparent color information from a **tRNS** chunk. This can be modified by a later call to `png_set_tRNS`.

9. 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_method);

    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_BASE
                   for PNG 1.0)

   filter_method  - (must be PNG_FILTER_TYPE_BASE
                   for PNG 1.0, and can also be
                   PNG_INTRAPIXEL_DIFFERENCING if
                   the PNG datastream is embedded in
                   a MNG-1.0 datastream)

   Any of width, height, color_type, bit_depth,
   interlace_type, compression_type, or filter_method can
   be NULL if you are not interested in their values.

   Note that png_get_IHDR() returns 32-bit data into
   the application's width and height variables.
   This is an unsafe situation if these are not png_uint_32
   variables.  In such situations, the
   png_get_image_width() and png_get_image_height()
   functions described below are safer.

   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);

   interlace_type   = png_get_interlace_type(png_ptr,
                        info_ptr);

   compression_type = png_get_compression_type(png_ptr,
                        info_ptr);

   filter_method    = png_get_filter_type(png_ptr,
                        info_ptr);

   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
                    This value, the bit_depth, color_type,
                    and the number of channels can change
                    if you use transforms such as
                    png_set_expand(). See
                    png_read_update_info(), below.
```

```
    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())).
```

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.

The colorspace data from **gAMA**, **cHRM**, **sRGB**, **sRGB**, and **sBIT** chunks is simply returned to give the application information about how the image was encoded. Libpng itself only does transformations using the file gamma when combining semitransparent pixels with the background color, and, since libpng-1.6.0, when converting between 8-bit sRGB and 16-bit linear pixels within the simplified API. Libpng also uses the file gamma when converting RGB to gray, beginning with libpng-1.0.5, if the application calls `png_set_rgb_to_gray()`).

```
    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, &file_gamma);
    png_get_gAMA_fixed(png_ptr, info_ptr, &int_file_gamma);

    file_gamma     - the gamma at which the file is
                     written (PNG_INFO_gAMA)

    int_file_gamma - 100,000 times the gamma at which the
                     file is written

    png_get_cHRM(png_ptr, info_ptr,  &white_x, &white_y, &red_x,
                     &red_y, &green_x, &green_y, &blue_x, &blue_y)
    png_get_cHRM_XYZ(png_ptr, info_ptr, &red_X, &red_Y, &red_Z,
                     &green_X, &green_Y, &green_Z, &blue_X, &blue_Y,
                     &blue_Z)
    png_get_cHRM_fixed(png_ptr, info_ptr, &int_white_x,
                     &int_white_y, &int_red_x, &int_red_y,
                     &int_green_x, &int_green_y, &int_blue_x,
                     &int_blue_y)
    png_get_cHRM_XYZ_fixed(png_ptr, info_ptr, &int_red_X, &int_red_Y,
                     &int_red_Z, &int_green_X, &int_green_Y,
                     &int_green_Z, &int_blue_X, &int_blue_Y,
                     &int_blue_Z)

    {white,red,green,blue}_{x,y}
                     A color space encoding specified using the
                     chromaticities of the end points and the
                     white point. (PNG_INFO_cHRM)
```

```
{red,green,blue}_{X,Y,Z}
                  A color space encoding specified using the
                  encoding end points - the CIE tristimulus
                  specification of the intended color of the red,
                  green and blue channels in the PNG RGB data.
                  The white point is simply the sum of the three
                  end points. (PNG_INFO_cHRM)

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_type - 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_alpha,
                  &num_trans, &trans_color);

trans_alpha     - array of alpha (transparency)
                  entries for palette (PNG_INFO_tRNS)

num_trans       - number of transparent entries
                  (PNG_INFO_tRNS)

trans_color     - graylevel or color sample values of
                  the single transparent color for
                  non-paletted images (PNG_INFO_tRNS)

png_get_eXIf_1(png_ptr, info_ptr, &num_exif, &exif);
                  (PNG_INFO_eXIf)

exif            - Exif profile (array of png_byte)

png_get_hIST(png_ptr, info_ptr, &hist);
                  (PNG_INFO_hIST)

hist            - histogram of palette (array of
                  png_uint_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 (of type
                 png_color_16p) (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.  Must contain
                    1-79 characters.

text_ptr[i].text  - text comments for current
                    keyword.  Can be empty.

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 (empty
                    string for unknown).

text_ptr[i].lang_key  - keyword in UTF-8
                    (empty string for unknown).

Note that the itxt_length, lang, and lang_key
members of the text_ptr structure only exist when the
library is built with iTXt chunk support.  Prior to
libpng-1.4.0 the library was built by default without
iTXt support. Also note that when iTXt is supported,
they contain NULL pointers when the "compression"
field contains PNG_TEXT_COMPRESSION_NONE or
PNG_TEXT_COMPRESSION_zTXt.

num_text       - number of comments (same as
                 num_comments; you can put NULL here
                 to avoid the duplication)

Note while png_set_text() will accept text, language,
and translated keywords that can be NULL pointers, the
structure returned by png_get_text will always contain
regular zero-terminated C strings.  They might be
empty strings but they will never be NULL pointers.

num_spalettes = png_get_sPLT(png_ptr, info_ptr,
   &palette_ptr);

num_spalettes  - number of sPLT chunks read.
```

```
palette_ptr    - array of palette structures holding
                 contents of one or more 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 (can be negative)

offset_y       - positive offset from the top edge
                 of the screen (can be negative)

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 (an integer)

width       - width of a pixel in physical scale units

height      - height of a pixel in physical scale units
              (width and height are doubles)

png_get_sCAL_s(png_ptr, info_ptr, &unit, &width,
   &height)

unit        - physical scale units (an integer)

width       - width of a pixel in physical scale units
              (expressed as a string)

height      - height of a pixel in physical scale units
              (width and height are strings like "2.54")

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's data

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

The value of "i" corresponds to the order in which the
chunks were read from the PNG file or inserted with the
png_set_unknown_chunks() function.
```

```
   The value of "location" is a bitwise "or" of

        PNG_HAVE_IHDR  (0x01)
        PNG_HAVE_PLTE  (0x02)
        PNG_AFTER_IDAT (0x08)
```

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)

   res_x = png_get_x_pixels_per_inch(png_ptr,
      info_ptr)

   res_y = png_get_y_pixels_per_inch(png_ptr,
      info_ptr)

   res_x_and_y = png_get_pixels_per_inch(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

   Note that because of the way the resolutions are
      stored internally, the inch conversions won't
      come out to exactly even number.  For example,
      72 dpi is stored as 0.28346 pixels/meter, and
      when this is retrieved it is 71.9988 dpi, so
      be sure to round the returned value appropriately
      if you want to display a reasonable-looking result.
```

The data from the **oFFs** chunk can be retrieved in several convenient forms:

```
   x_offset = png_get_x_offset_microns(png_ptr, info_ptr);

   y_offset = png_get_y_offset_microns(png_ptr, info_ptr);

   x_offset = png_get_x_offset_inches(png_ptr, info_ptr);

   y_offset = png_get_y_offset_inches(png_ptr, info_ptr);

   Each of these returns 0 [signifying "unknown" if both
      x and y are 0] if the data is not present or if the
      chunk is present but the unit is the pixel.  The
      remark about inexact inch conversions applies here
      as well, because a value in inches can't always be
      converted to microns and back without some loss
      of precision.
```

For more information, see 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*. The text string, language code, and translated keyword may be empty or NULL pointers. 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 mak*e 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().

10. 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.

Transformations you request are ignored if they don't have any meaning for a particular input data format. However some transformations can have an effect as a result of a previous transformation. If you specify a contradictory set of transformations, for example both adding and removing the alpha channel, you cannot predict the final result.

The color used for the transparency values should be supplied in the same format/depth as the current image data. It is stored in the same format/depth as the image data in a **tRNS** chunk, so this is what libpng expects for this data.

The color used for the background value depends on the need_expand argument as described 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() or png_set_add_alpha() 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_scale_16() is called to transform it to regular RGB RGB triplets, or png_set_filler() or png_set_add_alpha() is called to insert two filler bytes, either before or after each RRGGBB triplet. Similarly, *8-bit or 16-bit grayscale data* can be modified with png_set_filler(), png_set_add_alpha(), png_set_strip_16(), or png_set_scale_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)
        png_set_palette_to_rgb(png_ptr);

    if (png_get_valid(png_ptr, info_ptr,
        PNG_INFO_tRNS)) png_set_tRNS_to_alpha(png_ptr);

    if (color_type == PNG_COLOR_TYPE_GRAY &&
        bit_depth < 8) png_set_expand_gray_1_2_4_to_8(png_ptr);
```

The first two 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.

As of libpng version 1.2.9, `png_set_expand_gray_1_2_4_to_8()` was added. It expands the sample depth without changing **tRNS** to alpha.

As of libpng version 1.5.2, `png_set_expand_16()` was added. It behaves as `png_set_expand()`; however, the resultant channels have 16 bits rather than 8. Use this when the output color or gray channels are made linear to avoid fairly severe accuracy loss.

```
   if (bit_depth < 16)
      png_set_expand_16(png_ptr);
```

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)
#if PNG_LIBPNG_VER >= 10504
       png_set_scale_16(png_ptr);
#else
       png_set_strip_16(png_ptr);
#endif
```

(The more accurate "`png_set_scale_16()`" API became available in libpng version 1.5.4).

If you need to process the alpha channel on the image separately from the image data (for example if you convert it to a bitmap mask) it is possible to have libpng strip the channel leaving just RGB or gray data:

```
    if (color_type & PNG_COLOR_MASK_ALPHA)
       png_set_strip_alpha(png_ptr);
```

If you strip the alpha channel you need to find some other way of dealing with the information. If, instead, you want to convert the image to an opaque version with no alpha channel use `png_set_background`; see below.

As of libpng version 1.5.2, almost all useful expansions are supported, the major ommissions are conversion of grayscale to indexed images (which can be done trivially in the application) and conversion of indexed to grayscale (which can be done by a trivial manipulation of the palette.)

In the following table, the 01 means grayscale with depth<8, 31 means indexed with depth<8, other numerals represent the color type, "T" means the **tRNS** chunk is present, A means an alpha channel is present, and O means **tRNS** or *alpha* is present but all pixels in the image are *opaque*.

```
   FROM  01  31   0  0T  0O   2  2T  2O   3  3T  3O  4A  4O  6A  6O
    TO
    01    -  [G]  -   -   -   -   -   -   -   -   -   -   -   -   -
    31   [Q]  Q  [Q] [Q] [Q]  Q   Q   Q   Q   Q   Q  [Q] [Q]  Q   Q
     0    1   G   +   .   .   G   G   G   G   G   G   B   B  GB  GB
    0T    lt  Gt  t   +   .   Gt  G   G   Gt  G   G   Bt  Bt GBt GBt
    0O    lt  Gt  t   .   +   Gt  Gt  G   Gt  Gt  G   Bt  Bt GBt GBt
     2    C   P   C   C   C   +   .   .   C   -   -  CB  CB   B   B
    2T    Ct  -   Ct  C   C   t   +   t   -   -   -  CBt CBt  Bt  Bt
    2O    Ct  -   Ct  C   C   t   t   +   -   -   -  CBt CBt  Bt  Bt
     3   [Q]  p  [Q] [Q] [Q]  Q   Q   Q   +   .   .  [Q] [Q]  Q   Q
    3T   [Qt] p  [Qt][Q] [Q]  Qt  Qt  Qt  t   +   t  [Qt][Qt] Qt  Qt
    3O   [Qt] p  [Qt][Q] [Q]  Qt  Qt  Qt  t   t   +  [Qt][Qt] Qt  Qt
    4A    lA  G   A   T   T   GA  GT  GT  GA  GT  GT  +   BA  G  GBA
    4O    lA GBA  A   T   T   GA  GT  GT  GA  GT  GT  BA  +  GBA  G
    6A    CA  PA  CA  C   C   A   T  tT   PA  P   P   C  CBA  +   BA
    6O    CA PBA  CA  C   C   A  tT   T   PA  P   P  CBA  C   BA  +

Within the matrix,
     "+" identifies entries where 'from' and 'to' are the same.
     "-" means the transformation is not supported.
     "." means nothing is necessary (a tRNS chunk can just be ignored).
     "t" means the transformation is obtained by png_set_tRNS.
     "A" means the transformation is obtained by png_set_add_alpha().
     "X" means the transformation is obtained by png_set_expand().
     "1" means the transformation is obtained by
         png_set_expand_gray_1_2_4_to_8() (and by png_set_expand()
         if there is no transparency in the original or the final
         format).
     "C" means the transformation is obtained by png_set_gray_to_rgb().
     "G" means the transformation is obtained by png_set_rgb_to_gray().
     "P" means the transformation is obtained by
         png_set_expand_palette_to_rgb().
     "p" means the transformation is obtained by png_set_packing().
     "Q" means the transformation is obtained by png_set_quantize().
     "T" means the transformation is obtained by
         png_set_tRNS_to_alpha().
     "B" means the transformation is obtained by
         png_set_background(), or png_strip_alpha().
```

When an entry has multiple transforms listed all are required to cause the right overall transformation. When two transforms are separated by a comma either will do the job. When transforms are enclosed in [] the transform should do the job but this is currently unimplemented - a different format will result if the suggested transformations are used.

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_8p 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 or 6 bytes. This code expands them into 4 or 8 bytes for windowing systems that need them in this format:

```
    if (color_type == PNG_COLOR_TYPE_RGB)
       png_set_filler(png_ptr, filler, PNG_FILLER_BEFORE);
```

where "`filler`" is the 8-bit 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. When filling an 8-bit pixel, the least significant 8 bits of the number are used, if a 16-bit number is supplied. This transformation does not affect images that already have full alpha channels. To add an opaque alpha channel, use `filler=0xffff` and `PNG_FILLER_AFTER` which will generate `RGBA` pixels.

Note that `png_set_filler()` does not change the color type. If you want to do that, you can add a true alpha channel with

```
    if (color_type == PNG_COLOR_TYPE_RGB ||
       color_type == PNG_COLOR_TYPE_GRAY)
       png_set_add_alpha(png_ptr, filler, PNG_FILLER_AFTER);
```

where "`filler`" contains the alpha value to assign to each pixel. The `png_set_add_alpha()` function was added in libpng-1.2.7.

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(png_ptr, error_action,
          double red_weight, double 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

    green_weight:     weight of green component
                      If either weight is negative, default
                      weights are used.
```

In the corresponding fixed point API the `red_weight` and `green_weight` values are simply scaled by `100,000`:

```
    png_set_rgb_to_gray(png_ptr, error_action,
       png_fixed_point red_weight,
       png_fixed_point green_weight);
```

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. Background and **`sBIT`** data will be silently converted to grayscale, using the green channel data for **`sBIT`**, regardless of the `error_action` setting.

The default values come from the PNG file **`cHRM`** chunk if present; otherwise, the defaults correspond to the ITU-R recommendation 709, and also the sRGB color space, as recommended in the Charles Poynton's Colour FAQ, Copyright (c) 2006-11-28 Charles Poynton, in section 9:

<http://www.poynton.com/notes/colour_and_gamma/ColorFAQ.html#RTFToC9>

```
    Y = 0.2126 * R + 0.7152 * G + 0.0722 * B
```

Previous versions of this document, 1998 through 2002, recommended a slightly different formula:

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

Libpng uses an integer approximation:

```
    Y = (6968 * R + 23434 * G + 2366 * B)/32768
```

The calculation is done in a linear colorspace, if the image gamma can be determined.

The `png_set_background()` function has been described already; it tells libpng to composite images with alpha or simple transparency against the supplied background color. For compatibility with versions of libpng earlier than libpng-1.5.4 it is recommended that you call the function after reading the file header, even if you don't want to use the color in a **bKGD** chunk, if one exists.

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 how the color is represented, both the format of the component values in the color (the number of bits) and the gamma encoding of the color. The function takes two arguments, `background_gamma_mode` and `need_expand` to convey this information; however, only two combinations are likely to be useful:

```
    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/*needs to be expanded*/, 1);
    else
       png_set_background(png_ptr, &my_background,
           PNG_BACKGROUND_GAMMA_SCREEN, 0/*do not expand*/, 1);
```

The second call was described above - `my_background` is in the format of the final, display, output produced by libpng. Because you now know the format of the PNG it is possible to avoid the need to choose either 8-bit or 16-bit output and to retain palette images (the palette colors will be modified appropriately and the **tRNS** chunk removed.) However, if you are doing this, take great care not to ask for transformations without checking first that they apply!

In the first call the background color has the original bit depth and color type of the PNG file. So, for palette images the color is supplied as a palette index and for low bit greyscale images the color is a reduced bit value in `image_background->gray`.

If you didn't call `png_set_gamma()` before reading the file header, for example if you need your code to remain compatible with older versions of libpng prior to libpng-1.5.4, this is the place to call it.

Do not call it if you called `png_set_alpha_mode()`; doing so will damage the settings put in place by `png_set_alpha_mode()`. (If `png_set_alpha_mode()` is supported then you can certainly do `png_set_gamma()` before reading the PNG header.)

This API unconditionally sets the screen and file gamma values, so it will override the value in the PNG file unless it is called before the PNG file reading starts. For this reason you must always call it with the PNG file value when you call it in this position:

```
   if (png_get_gAMA(png_ptr, info_ptr, &file_gamma))
      png_set_gamma(png_ptr, screen_gamma, file_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 than will fit on your screen, `png_set_quantize()` will do that. Note that this is a simple match quantization that merely finds the closest color available. This should work fairly well with optimized palettes, but fairly badly with linear color cubes. If you pass a palette that is larger than `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, libpng 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_uint_16p histogram = NULL;

         png_get_hIST(png_ptr, info_ptr,
             &histogram);
         png_set_quantize(png_ptr, palette, num_palette,
            max_screen_colors, histogram, 1);
      }

      else
      {
         png_color std_color_cube[MAX_SCREEN_COLORS] =
            { ... colors ... };

         png_set_quantize(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_TYPE_GRAY)
      png_set_invert_mono(png_ptr);
```

This function can also be used to invert grayscale and gray-alpha images:

```
   if (color_type == PNG_COLOR_TYPE_GRAY ||
       color_type == PNG_COLOR_TYPE_GRAY_ALPHA)
      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_structp png_ptr, png_row_infop
        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. Take care with interlaced images if you do the interlace yourself - the width of the row is the width in '`row_info`', not the overall image width.

If supported, libpng provides two information routines that you can use to find where you are in processing the image:

```
   png_get_current_pass_number(png_structp png_ptr);
   png_get_current_row_number(png_structp png_ptr);
```

Don't try using these outside a *transform callback* - firstly they are only supported if user transforms are supported, secondly they may well *return unexpected results* unless the row is actually being processed at the moment they are called.

With interlaced images the value returned is the row in the input sub-image image. Use `PNG_ROW_FROM_PASS_ROW(row, pass)` and `PNG_COL_FROM_PASS_COL(col, pass)` to find the output pixel (x,y) given an interlaced sub-image pixel (row,col,pass).

The discussion of interlace handling above contains more information on how to use these values.

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.

```
    png_read_update_info(png_ptr, info_ptr);
```

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. You may only call `png_read_update_info()` once with a particular `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.

Be sure that your platform can allocate the buffer that you'll need. libpng internally checks for oversize width, but you'll need to do your own check for `number_of_rows*width*pixel_size` if you are using a multiple-row buffer:

```
   /* Guard against integer overflow */
   if (number_of_rows > PNG_SIZE_MAX/(width*pixel_size)) {
        png_error(png_ptr,"image_data buffer would be too large");
```

}

Remember: Before you call `png_read_update_info()`, the `png_get_*()` functions return the values corresponding to the original PNG image. After you call `png_read_update_info` the values refer to the image that libpng will output. Consequently you must call all the `png_set_` functions before you call `png_read_update_info()`. This is particularly important for `png_set_interlace_handling()` - if you are going to call `png_read_update_info()` you must call `png_set_interlace_handling()` before it unless you want to receive interlaced output.

11. 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()` (unless you call `png_read_update_info()`) 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_pointer, NULL);
```

If the file is interlaced (`interlace_type != 0` in the **`IHDR`** chunk), things get somewhat harder. The only current (PNG Specification version 1.2) interlacing type for PNG is (`interlace_type == PNG_INTERLACE_ADAM7`); 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. This number is defined (from libpng 1.5) as `PNG_INTERLACE_ADAM7_PASSES` in `png.h`

libpng can fill out those images or it can give them to you "as is". It is almost always better to have libpng handle the interlacing for you. If you want the images 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, as is likely, 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. You then need to read the whole image '`number_of_passes`' times. Each time will distribute the pixels from the current pass to the correct place in the output image, so you need to supply the same rows to `png_read_rows` in each 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_row()` or `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);
    or
    png_read_row(png_ptr, row_pointers, NULL);
```

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);
    or
    png_read_row(png_ptr, NULL, row_pointers);
```

If you don't want libpng to handle the interlacing details, just call `png_read_rows()` `PNG_INTERLACE_ADAM7_PASSES` times to read in all the images. Each of the images is a valid image by itself; however, you will almost certainly need to distribute the pixels from each sub-image to the correct place. This is where everything gets very tricky.

If you want to retrieve the separate images you must pass the correct number of rows to each successive call of `png_read_rows()`. The calculation gets pretty complicated for small images, where some sub-images may not even exist because either their width or height ends up *zero*. libpng provides two macros to help you in 1.5 and later versions:

```
   png_uint_32 width = PNG_PASS_COLS(image_width, pass_number);
   png_uint_32 height = PNG_PASS_ROWS(image_height, pass_number);
```

Respectively these tell you the width and height of the sub-image corresponding to the numbered pass. '`pass`' is in in the range `0` to `6` - this can be confusing because the specification refers to the same passes as `1` to `7`! Be careful, you must check both the width and height before calling `png_read_rows()` and not call it for that pass *if either* is `zero`.

You can, of course, read each sub-image row by row. If you want to produce optimal code to make a pixel-by-pixel transformation of an interlaced image this is the best approach; read each row of each pass, transform it, and write it out to a new interlaced image.

If you want to de-interlace the image yourself libpng provides further macros to help that tell you where to place the pixels in the output image. Because the interlacing scheme is rectangular - sub-image pixels are always arranged on a rectangular grid - all you need to know for each pass is the starting column and row in the output image of the first pixel plus the spacing between each pixel. As of libpng 1.5 there are four macros to retrieve this information:

```
   png_uint_32 x = PNG_PASS_START_COL(pass);
   png_uint_32 y = PNG_PASS_START_ROW(pass);
   png_uint_32 xStep = 1U << PNG_PASS_COL_SHIFT(pass);
   png_uint_32 yStep = 1U << PNG_PASS_ROW_SHIFT(pass);
```

These allow you to write the obvious loop:

```
   png_uint_32 input_y = 0;
   png_uint_32 output_y = PNG_PASS_START_ROW(pass);

   while (output_y < output_image_height)
   {
      png_uint_32 input_x = 0;
      png_uint_32 output_x = PNG_PASS_START_COL(pass);

      while (output_x < output_image_width)
      {
         image[output_y][output_x] =
             subimage[pass][input_y][input_x++];

         output_x += xStep;
      }

      ++input_y;
      output_y += yStep;
   }
```

Notice that the steps between successive output rows and columns are returned as shifts. This is possible because the pixels in the subimages are always a *power of 2* apart - *1, 2, 4 or 8 pixels* - in the original image. In practice you may need to directly calculate the output coordinate given an input coordinate. libpng provides two further macros for this purpose:

```
   png_uint_32 output_x = PNG_COL_FROM_PASS_COL(input_x, pass);
   png_uint_32 output_y = PNG_ROW_FROM_PASS_ROW(input_y, pass);
```

Finally a pair of macros are provided to tell you if a particular image row or column appears in a given pass:

```
   int col_in_pass = PNG_COL_IN_INTERLACE_PASS(output_x, pass);
   int row_in_pass = PNG_ROW_IN_INTERLACE_PASS(output_y, pass);
```

Bear in mind that you will probably also need to check the width and height of the pass in addition to the above to be sure the pass even exists!

With any luck you are convinced by now that you don't want to do your own interlace handling. In reality normally the only good reason for doing this is if you are processing PNG files on a pixel-by-pixel basis and don't want to load the whole file into memory when it is interlaced.

libpng includes a test program, `pngvalid`, that illustrates reading and writing of interlaced images. If you can't get interlacing to work in your code and don't want to leave it to libpng (the recommended approach), see how `pngvalid.c` does it.

12. Finishing a sequential read

After you are finished reading the image through the low-level interface, you can finish reading the file.

If you want to use a different crc action for handling *CRC errors* in chunks after the image data, you can call `png_set_crc_action()` again at this point.

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.

```
    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 ERROR;
    }

   png_read_end(png_ptr, end_info);
```

If you are not interested, you should still call `png_read_end()` but you can pass `NULL`, avoiding the need to create an `end_info` structure. If you do this, libpng will not process any chunks after **`IDAT`** other than skipping over them and perhaps (depending on whether you have called `png_set_crc_action`) checking their *CRC*s while looking for the **`IEND`** chunk.

```
   png_read_end(png_ptr, (png_infop)NULL);
```

If you don't call `png_read_end()`, then your *file pointer* will be left pointing to the first chunk after the last **`IDAT`**, which is probably not what you want if you expect to read something beyond the end of the PNG datastream.

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

```
   png_destroy_read_struct(&png_ptr, &info_ptr,
       &end_info);
```

or, if you didn't create an `end_info` structure,

```
   png_destroy_read_struct(&png_ptr, &info_ptr,
```

```
      (png_infopp)NULL);
```

It is also possible to individually free the info_ptr members that point to libpng-allocated storage with the following function:

```
    png_free_data(png_ptr, info_ptr, mask, seq)

    mask - identifies data to be freed, a mask
           containing the bitwise OR of one or
           more of
             PNG_FREE_PLTE, PNG_FREE_TRNS,
             PNG_FREE_HIST, PNG_FREE_ICCP,
             PNG_FREE_PCAL, PNG_FREE_ROWS,
             PNG_FREE_SCAL, PNG_FREE_SPLT,
             PNG_FREE_TEXT, PNG_FREE_UNKN,
           or simply PNG_FREE_ALL

    seq  - sequence number of item to be freed
           (-1 for all items)
```

This function may be safely called when the relevant storage has already been freed, or has not yet been allocated, or was allocated by the user and not by libpng, and will in those cases do nothing. The "seq" parameter is ignored if only one item of the selected data type, such as **PLTE**, is allowed. If "seq" 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 in the structure is freed, where n is "seq".

The default behavior is only to free data that was allocated internally by libpng. This can be changed, so that libpng will not free the data, or so that it will free data that was allocated by the user with png_malloc() or png_calloc() and passed in via a png_set_*() function, with

```
    png_data_freer(png_ptr, info_ptr, freer, mask)

    freer  - one of
               PNG_DESTROY_WILL_FREE_DATA
               PNG_SET_WILL_FREE_DATA
               PNG_USER_WILL_FREE_DATA

    mask   - which data elements are affected
             same choices as in png_free_data()
```

This function only affects data that has already been allocated. You can call this function after reading the PNG data but before calling any png_set_*() functions, to control whether the user or the png_set_*() function is responsible for freeing any existing data that might be present, and again after the png_set_*() functions to control whether the user or png_destroy_*() is supposed to free the data. When the user assumes responsibility for libpng-allocated data, the application must use png_free() to free it, and when the user transfers responsibility to libpng for data that the user has allocated, the user must have used png_malloc() or png_calloc() to allocate it.

If you allocated your row_pointers in a single block, as suggested above in the description of the high level read interface, you must not transfer responsibility for freeing it to the png_set_rows or png_read_destroy function, because they would also try to free the individual row_pointers[i].

If you allocated text_ptr.text, text_ptr.lang, and text_ptr.translated_keyword separately, do not transfer responsibility for freeing text_ptr to libpng, because when libpng fills a png_text structure

it combines these members with the key member, and png_free_data() will free only text_ptr.key. Similarly, if you transfer responsibility for free'ing text_ptr from libpng to your application, your application must not separately free those members.

The png_free_data() function will turn off the "valid" flag for anything it frees. If you need to turn the flag off for a chunk that was freed by your application instead of by libpng, you can use

```
    png_set_invalid(png_ptr, info_ptr, mask);

    mask - identifies the chunks to be made invalid,
           containing the bitwise OR of one or
           more of
             PNG_INFO_gAMA, PNG_INFO_sBIT,
             PNG_INFO_cHRM, PNG_INFO_PLTE,
             PNG_INFO_tRNS, PNG_INFO_bKGD,
             PNG_INFO_eXIf,
             PNG_INFO_hIST, PNG_INFO_pHYs,
             PNG_INFO_oFFs, PNG_INFO_tIME,
             PNG_INFO_pCAL, PNG_INFO_sRGB,
             PNG_INFO_iCCP, PNG_INFO_sPLT,
             PNG_INFO_sCAL, PNG_INFO_IDAT
```

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

13. Reading PNG files progressively

The progressive reader is slightly different from 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 ERROR;

    info_ptr = png_create_info_struct(png_ptr);

    if (!info_ptr)
    {
       png_destroy_read_struct(&png_ptr,
          (png_infopp)NULL, (png_infopp)NULL);
       return ERROR;
```

```
   }

   if (setjmp(png_jmpbuf(png_ptr)))
   {
      png_destroy_read_struct(&png_ptr, &info_ptr,
         (png_infopp)NULL);
      return ERROR;
   }

   /* 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 ERROR;
   }

   /* 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 than 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);

   /* At this point you can call png_process_data_skip if
      you want to handle data the library will skip yourself;
      it simply returns the number of bytes to skip (and stops
      libpng skipping that number of bytes on the next
      png_process_data call).
   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 is where you turn on interlace handling,
      assuming you don't want to do it yourself.

      If you need to you can stop the processing of
      your original input data at this point by calling
      png_process_data_pause.  This returns the number
      of unprocessed bytes from the last png_process_data
      call - it is up to you to ensure that the next call
      sees these bytes again.  If you don't want to bother
      with this you can get libpng to cache the unread
      bytes by setting the 'save' parameter (see png.h) but
      then libpng will have to copy the data internally.
    */
}

/* 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.

      If you did not turn on interlace handling then
      the callback is called for each row of each
      sub-image when the image is interlaced.  In this
      case 'row_num' is the row in the sub-image, not
      the row in the output image as it is in all other
      cases.

      For the non-NULL rows of interlaced images when
      you have switched on libpng interlace handling,
      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 if you switch on interlace handling;
```

```
     */

        png_progressive_combine_row(png_ptr, old_row,
          new_row);

    /* where old_row is what was displayed
       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.

       You can also call png_process_data_pause in this
       callback - see above.
    */
}

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.

1. 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 ERROR;
```

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 ERROR;

    png_infop info_ptr = png_create_info_struct(png_ptr);
    if (!info_ptr)
    {
       png_destroy_write_struct(&png_ptr,
           (png_infopp)NULL);
       return ERROR;
    }
```

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_write_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 ERROR;
    }
    ...
    return;
```

If you would rather avoid the complexity of setjmp/longjmp issues, you can compile libpng with PNG_NO_SETJMP, in which case *errors* will result in a call to PNG_ABORT() which defaults to abort().

You can #define PNG_ABORT() to a function that does something more useful than abort(), as long as *your function does not return*.

Checking for invalid palette index on write was added at libpng 1.5.10. If a pixel contains an invalid (out-of-range) index libpng issues a *benign error*. This is enabled by default because this condition is an *error* according to the PNG specification, Clause 11.3.2, but the *error* can be ignored in each png_ptr with

```
   png_set_check_for_invalid_index(png_ptr, 0);
```

If the *error* is ignored, or if `png_benign_error()` treats it as a *warning*, any invalid pixels are written as-is by the encoder, resulting in an invalid PNG datastream as output. In this case the application is responsible for ensuring that the pixel indexes are in range when it writes a **`PLTE`** chunk with fewer entries than the bit depth would allow.

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);
```

If you are embedding your PNG into a datastream such as MNG, and don't want libpng to write the 8-byte signature, or if you have already written the signature in your application, use

```
   png_set_sig_bytes(png_ptr, 8);
```

to inform libpng that it should not write a signature.

2. 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_structp 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);
```

When this function is called the row has already been completely processed and it has also been written out. The '`row`' and '`pass`' refer to the next row to be handled. For the non-interlaced case the row that was just handled is simply one less than the passed in row number, and pass will always be 0. For the interlaced case the same applies unless the row value is 0, in which case the row just handled was the last one from one of the preceding passes. Because interlacing may skip a pass you cannot be sure that the preceding pass is just 'pass-1', if you really need to know what the last pass is record (row,pass) from the *callback* and use the last recorded value each time.

As with the user transform you can find the output row using the `PNG_ROW_FROM_PASS_ROW` macro.

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 values are `0` (as of the July 1999 PNG specification, version 1.2) or `64` (if you are writing a PNG datastream that is to be embedded in a MNG datastream). The third parameter is a flag that indicates which filter type(s) are to be tested for each scanline. See the PNG specification for details on the specific filter types.

```
    /* turn on or off filtering, and/or choose
       specific filters.  You can use either a single
       PNG_FILTER_VALUE_NAME or the bitwise OR of one
       or more PNG_FILTER_NAME masks.
     */
    png_set_filter(png_ptr, 0,
       PNG_FILTER_NONE  | PNG_FILTER_VALUE_NONE |
       PNG_FILTER_SUB   | PNG_FILTER_VALUE_SUB  |
       PNG_FILTER_UP    | PNG_FILTER_VALUE_UP   |
       PNG_FILTER_AVG   | PNG_FILTER_VALUE_AVG  |
       PNG_FILTER_PAETH | PNG_FILTER_VALUE_PAETH|
       PNG_ALL_FILTERS  | PNG_FAST_FILTERS);
```

If an application wants to start and stop using particular filters during compression, it should start out with all of the filters (to ensure that the previous row of pixels will be stored in case it's needed later), and then add and remove them after the start of compression.

If you are writing a PNG datastream that is to be embedded in a MNG datastream, the second parameter can be either 0 or 64.

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 (`zlib.h` and `algorithm.txt`, distributed with zlib) for details on the compression levels.

```
    #include zlib.h

    /* Set the zlib compression level */
    png_set_compression_level(png_ptr,
        Z_BEST_COMPRESSION);

    /* Set other zlib parameters for compressing IDAT */
    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);
    png_set_compression_buffer_size(png_ptr, 8192)

    /* Set zlib parameters for text compression
     * If you don't call these, the parameters
     * fall back on those defined for IDAT chunks
     */
```

```
    png_set_text_compression_mem_level(png_ptr, 8);
    png_set_text_compression_strategy(png_ptr,
        Z_DEFAULT_STRATEGY);
    png_set_text_compression_window_bits(png_ptr, 15);
    png_set_text_compression_method(png_ptr, 8);
```

3. 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_method)

    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_method  - (must be PNG_FILTER_TYPE_DEFAULT
```

```
                      or, if you are writing a PNG to
                      be embedded in a MNG datastream,
                      can also be
                      PNG_INTRAPIXEL_DIFFERENCING)
```

If you call `png_set_IHDR()`, the call must appear before any of the other `png_set_*()` functions, because they might require access to some of the **`IHDR`** settings. The remaining `png_set_*()` functions can be called in any order.

If you wish, you can reset the `compression_type`, `interlace_type`, or `filter_method` later by calling `png_set_IHDR()` again; if you do this, the `width`, `height`, `bit_depth`, and `color_type` must be the same in each call.

```
    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, file_gamma);
    png_set_gAMA_fixed(png_ptr, info_ptr, int_file_gamma);

    file_gamma     - the gamma at which the image was
                     created (PNG_INFO_gAMA)

    int_file_gamma - 100,000 times the gamma at which
                     the image was created

    png_set_cHRM(png_ptr, info_ptr,  white_x, white_y, red_x, red_y,
                     green_x, green_y, blue_x, blue_y)
    png_set_cHRM_XYZ(png_ptr, info_ptr, red_X, red_Y, red_Z, green_X,
                     green_Y, green_Z, blue_X, blue_Y, blue_Z)
    png_set_cHRM_fixed(png_ptr, info_ptr, int_white_x, int_white_y,
                     int_red_x, int_red_y, int_green_x, int_green_y,
                     int_blue_x, int_blue_y)
    png_set_cHRM_XYZ_fixed(png_ptr, info_ptr, int_red_X, int_red_Y,
                     int_red_Z, int_green_X, int_green_Y, int_green_Z,
                     int_blue_X, int_blue_Y, int_blue_Z)

    {white,red,green,blue}_{x,y}
                     A color space encoding specified using the chromaticities
                     of the end points and the white point.

    {red,green,blue}_{X,Y,Z}
                     A color space encoding specified using the encoding end
                     points - the CIE tristimulus specification of the
intended
                     color of the red, green and blue channels in the PNG RGB
                     data.  The white point is simply the sum of the three end
                     points.

    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_type - 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_alpha,
       num_trans, trans_color);

    trans_alpha    - array of alpha (transparency)
                     entries for palette (PNG_INFO_tRNS)

    num_trans      - number of transparent entries
                     (PNG_INFO_tRNS)

    trans_color    - graylevel or color sample values
                     (in order red, green, blue) of the
                     single transparent color for
                     non-paletted images (PNG_INFO_tRNS)

    png_set_eXIf_1(png_ptr, info_ptr, num_exif, exif);

    exif           - Exif profile (array of
                     png_byte) (PNG_INFO_eXIf)

    png_set_hIST(png_ptr, info_ptr, hist);
```

```
    hist           - histogram of palette (array of
                     png_uint_16) (PNG_INFO_hIST)

    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 (of type
                     png_color_16p) (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.  Must contain
                  1-79 characters.
    text_ptr[i].text  - text comments for current
                        keyword.  Can be NULL or empty.
    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 or
                        empty for unknown).
    text_ptr[i].translated_keyword  - keyword in UTF-8 (NULL
                        or empty for unknown).

    Note that the itxt_length, lang, and lang_key
    members of the text_ptr structure only exist when the
    library is built with iTXt chunk support.  Prior to
    libpng-1.4.0 the library was built by default without
    iTXt support. Also note that when iTXt is supported,
    they contain NULL pointers when the "compression"
    field contains PNG_TEXT_COMPRESSION_NONE or
    PNG_TEXT_COMPRESSION_zTXt.

    num_text       - number of comments

    png_set_sPLT(png_ptr, info_ptr, &palette_ptr,
       num_spalettes);

    palette_ptr    - array of png_sPLT_struct 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 (an integer)

    width       - width of a pixel in physical scale units

    height      - height of a pixel in physical scale units
                  (width and height are doubles)

    png_set_sCAL_s(png_ptr, info_ptr, unit, width, height)

    unit        - physical scale units (an integer)

    width       - width of a pixel in physical scale units
                  expressed as a string

    height      - height of a pixel in physical scale units
                 (width and height are strings like "2.54")

    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's data
    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`. Within each of the "`locations`", the chunks are sequenced according to their *position in the structure* (that is, the value of "`i`", which is the order in which the chunk was either read from the input file or defined with `png_set_unknown_chunks`).

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. 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 **`tEXt`** and **`zTXt`** chunks don't have a

language field, if you specify PNG_TEXT_COMPRESSION_NONE or PNG_TEXT_COMPRESSION_zTXt any language code or translated keyword will not be written out.

Until text gets around a few hundred 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 provided, 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_buffer(buffer, png_timep) is provided to convert from PNG time to an RFC 1123 format string. The caller must provide a writeable buffer of at least 29 bytes.

4. Writing unknown chunks

You can use the png_set_unknown_chunks function to queue up private chunks for writing. You give it a chunk name, location, raw data, and a size. You also must use png_set_keep_unknown_chunks() to ensure that libpng will handle them. 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, depending upon the specified location. 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.

Here is an example of writing two private chunks, prVt and miNE:

```
    #ifdef PNG_WRITE_UNKNOWN_CHUNKS_SUPPORTED
    /* Set unknown chunk data */
    png_unknown_chunk unk_chunk[2];
    strcpy((char *) unk_chunk[0].name, "prVt";
    unk_chunk[0].data = (unsigned char *) "PRIVATE DATA";
    unk_chunk[0].size = strlen(unk_chunk[0].data)+1;
    unk_chunk[0].location = PNG_HAVE_IHDR;
    strcpy((char *) unk_chunk[1].name, "miNE";
    unk_chunk[1].data = (unsigned char *) "MY CHUNK DATA";
    unk_chunk[1].size = strlen(unk_chunk[0].data)+1;
    unk_chunk[1].location = PNG_AFTER_IDAT;
    png_set_unknown_chunks(write_ptr, write_info_ptr,
        unk_chunk, 2);
    /* Needed because miNE is not safe-to-copy */
    png_set_keep_unknown_chunks(png, PNG_HANDLE_CHUNK_ALWAYS,
       (png_bytep) "miNE", 1);
    # if PNG_LIBPNG_VER < 10600
      /* Deal with unknown chunk location bug in 1.5.x and earlier */
      png_set_unknown_chunk_location(png, info, 0, PNG_HAVE_IHDR);
      png_set_unknown_chunk_location(png, info, 1, PNG_AFTER_IDAT);
    # endif
    # if PNG_LIBPNG_VER < 10500
      /* PNG_AFTER_IDAT writes two copies of the chunk prior to libpng-1.5.0,
       * one before IDAT and another after IDAT, so don't use it; only use
       * PNG_HAVE_IHDR location.  This call resets the location previously
       * set by assignment and png_set_unknown_chunk_location() for chunk 1.
       */
      png_set_unknown_chunk_location(png, info, 1, PNG_HAVE_IHDR);
    # endif
    #endif
```

5. 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 in 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 (deprecated).
    PNG_TRANSFORM_STRIP_FILLER_BEFORE Strip out leading
                                      filler bytes
    PNG_TRANSFORM_STRIP_FILLER_AFTER  Strip out trailing
                                      filler bytes
```

If you have valid image data in the info structure (you can use `png_set_rows()` to put image data in the info structure), simply do this:

```
    png_write_png(png_ptr, info_ptr, png_transforms, NULL)
```

where `png_transforms` is an integer containing the bitwise `OR` of some set of transformation flags. This call is equivalent to `png_write_info()`, followed the set of transformations indicated by the transform mask, then `png_write_image()`, and finally `png_write_end()`.

(The final parameter of this call is not yet used. Someday it might point to transformation parameters required by some future output transform.)

You must use `png_transforms` and not call any `png_set_transform()` functions when you use `png_write_png()`.

6. 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 strip 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 pixel 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 recover 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_structp png_ptr, png_row_infop
       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. If supported libpng also supplies an information routine that may be called from your *callback*:

```
   png_get_current_row_number(png_ptr);
   png_get_current_pass_number(png_ptr);
```

This returns the current row passed to the transform. With interlaced images the value returned is the row in the input sub-image image. Use `PNG_ROW_FROM_PASS_ROW(row, pass)` and `PNG_COL_FROM_PASS_COL(col, pass)` to find the output pixel (x,y) given an interlaced sub-image pixel (row,col,pass).

The discussion of interlace handling above contains more information on how to use these values.

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.

7. 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 the 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 the sub-images (`png_set_interlace_handling()` returns the number of 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);
```

Think carefully before you write an interlaced image. Typically code that reads such images reads all the image data into memory, uncompressed, before doing any processing. Only code that can display an image on the fly can take advantage of the interlacing and even then the image has to be exactly the correct size for the output device, because scaling an image requires adjacent pixels and these are not available until all the passes have been read.

If you do write an interlaced image you will hardly ever need to handle the interlacing yourself. Call `png_set_interlace_handling()` and use the approach described above.

The only time it is conceivable that you will really need to write an interlaced image pass-by-pass is when you have read one pass by pass and made some pixel-by-pixel transformation to it, as described in the read code above. In this case use the `PNG_PASS_ROWS` and `PNG_PASS_COLS` macros to determine the size of each sub-image in turn and simply write the rows you obtained from the read code.

8. 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 function:

```
    png_free_data(png_ptr, info_ptr, mask, seq)

    mask  - identifies data to be freed, a mask
            containing the bitwise OR of one or
            more of
              PNG_FREE_PLTE, PNG_FREE_TRNS,
              PNG_FREE_HIST, PNG_FREE_ICCP,
              PNG_FREE_PCAL, PNG_FREE_ROWS,
              PNG_FREE_SCAL, PNG_FREE_SPLT,
              PNG_FREE_TEXT, PNG_FREE_UNKN,
            or simply PNG_FREE_ALL

    seq   - sequence number of item to be freed
            (-1 for all items)
```

This function may be safely called when the relevant storage has already been freed, or has not yet been allocated, or was allocated by the user and not by libpng, and will in those cases do nothing. The "`seq`" parameter is ignored if only one item of the selected data type, such as **`PLTE`**, is allowed. If "`seq`" 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 in the structure is freed, where n is "`seq`".

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()`.

The default behavior is only to free data that was allocated internally by libpng. This can be changed, so that libpng will not free the data, or so that it will free data that was allocated by the user with `png_malloc()` or `png_calloc()` and passed in via a `png_set_*()` function, with

```
    png_data_freer(png_ptr, info_ptr, freer, mask)

    freer  - one of
               PNG_DESTROY_WILL_FREE_DATA
               PNG_SET_WILL_FREE_DATA
               PNG_USER_WILL_FREE_DATA

    mask   - which data elements are affected
             same choices as in png_free_data()
```

For example, to transfer responsibility for some data from a *read structure* to a *write structure*, you could use

```
    png_data_freer(read_ptr, read_info_ptr,
       PNG_USER_WILL_FREE_DATA,
       PNG_FREE_PLTE|PNG_FREE_tRNS|PNG_FREE_hIST)

    png_data_freer(write_ptr, write_info_ptr,
       PNG_DESTROY_WILL_FREE_DATA,
       PNG_FREE_PLTE|PNG_FREE_tRNS|PNG_FREE_hIST)
```

thereby briefly reassigning responsibility for freeing to the user but immediately afterwards reassigning it once more to the write_destroy function. Having done this, it would then be safe to destroy the *read structure* and continue to use the **`PLTE`**, **`tRNS`**, and **`hIST`** data in the *write structure*.

This function only affects data that has already been allocated. You can call this function before calling after the `png_set_*()` functions to control whether the user or `png_destroy_*()` is supposed to free the data. When the user assumes responsibility for libpng-allocated data, the application must use `png_free()` to free it, and when the user transfers responsibility to libpng for data that the user has allocated, the user must have used `png_malloc()` or `png_calloc()` to allocate it.

If you allocated `text_ptr.text`, `text_ptr.lang`, and `text_ptr.translated_keyword` separately, do not transfer responsibility for freeing `text_ptr` to libpng, because when libpng fills a `png_text` structure it combines these members with the key member, and `png_free_data()` will free only `text_ptr.key`. Similarly, if you transfer responsibility for free'ing `text_ptr` from libpng to your application, your application must not separately free those members. For a more compact example of writing a PNG image, see the file `example.c`.

V. Simplified API

The simplified API, which became available in libpng-1.6.0, hides the details of both libpng and the PNG file format itself. It allows PNG files to be read into a very limited number of in-memory bitmap formats or to be written from the same formats. If these formats do not accommodate your needs then you can, and should, use the more sophisticated APIs above - these support a wide variety of in-memory formats

and a wide variety of sophisticated transformations to those formats as well as a wide variety of APIs to manipulate ancillary information.

1. To read a PNG file using the simplified API:

1) Declare a '`png_image`' structure (see below) on the stack, set the version field to `PNG_IMAGE_VERSION` and the '`opaque`' pointer to `NULL` (this is REQUIRED, your program may crash if you don't do it.)

2) Call the appropriate `png_image_begin_read...` function.

3) Set the `png_image` '`format`' member to the required sample format.

4) Allocate a buffer for the image and, if required, the color-map.

5) Call `png_image_finish_read` to read the image and, if required, the color-map into your buffers.

There are no restrictions on the format of the PNG input itself; all valid color types, bit depths, and interlace methods are acceptable, and the input image is transformed as necessary to the requested in-memory format during the `png_image_finish_read()` step. The only caveat is that if you request a color-mapped image from a PNG that is full-color or makes complex use of an alpha channel the transformation is extremely lossy and the result may look terrible.

2. To write a PNG file using the simplified API:

1) Declare a '`png_image`' structure on the stack and `memset()` it to all *zero*.

2) Initialize the members of the structure that describe the image, setting the '`format`' member to the format of the image samples.

3) Call the appropriate `png_image_write...` function with a *pointer to the image* and, if necessary, the color-map to write the PNG data.

`png_image` is a structure that describes the in-memory format of an image when it is being read or defines the in-memory format of an image that you need to write. The "`png_image`" structure contains the following members:

```
    png_controlp opaque  Initialize to NULL, free with png_image_free
    png_uint_32  version Set to PNG_IMAGE_VERSION
    png_uint_32  width   Image width in pixels (columns)
    png_uint_32  height  Image height in pixels (rows)
    png_uint_32  format  Image format as defined below
    png_uint_32  flags   A bit mask containing informational flags
    png_uint_32  colormap_entries; Number of entries in the color-map
    png_uint_32  warning_or_error;
    char         message[64];
```

In the event of an *error* or *warning* the "`warning_or_error`" field will be set to a *non-zero value* and the '`message`' field will contain a `'\0'` terminated string with the libpng *error* or *warning message*. If both

warnings and an *error* were encountered, only the *error* is recorded. If there are multiple *warnings*, only the first one is recorded.

The *upper 30 bits* of the "warning_or_error" value *are reserved*; the *low two bits* contain a *two bit code* such that a value more than 1 indicates a failure in the API just called:

```
0 - no warning or error
1 - warning
2 - error
3 - error preceded by warning
```

The pixels (samples) of the image have one to four channels whose components have original values in the range 0 to 1.0:

```
1: A single gray or luminance channel (G).
2: A gray/luminance channel and an alpha channel (GA).
3: Three red, green, blue color channels (RGB).
4: Three color channels and an alpha channel (RGBA).
```

The channels are encoded in one of two ways:

a) As a small integer, value 0..255, contained in a single byte. For the alpha channel the original value is simply value/255. For the color or luminance channels the value is encoded according to the sRGB specification and matches the 8-bit format expected by typical display devices.

The color/gray channels are not scaled (pre-multiplied) by the alpha channel and are suitable for passing to color management software.

b) As a value in the range 0..65535, contained in a 2-byte integer, in the native byte order of the platform on which the application is running. All channels can be converted to the original value by dividing by 65535; all channels are linear. Color channels use the RGB encoding (RGB end-points) of the sRGB specification. This encoding is identified by the PNG_FORMAT_FLAG_LINEAR flag below.

When the simplified API needs to convert between sRGB and linear colorspaces, the actual sRGB transfer curve defined in the sRGB specification (see the article at https://en.wikipedia.org/wiki/SRGB) is used, *not the gamma=1/2.2 approximation* used elsewhere in libpng.

When an alpha channel is present it is expected to denote pixel coverage of the color or luminance channels and is returned as an associated alpha channel: the color/gray channels are scaled (pre-multiplied) by the alpha value.

The samples are either contained directly in the image data, between 1 and 8 bytes per pixel according to the encoding, or are held in a color-map indexed by bytes in the image data. In the case of a color-map the color-map entries are individual samples, encoded as above, and the image data has one byte per pixel to select the relevant sample from the color-map.

3. PNG_FORMAT_*

The #defines to be used in png_image::format. Each #define identifies a particular layout of channel data and, if present, alpha values. There are separate defines for each of the two component encodings.

A format is built up using single bit flag values. All combinations are valid. Formats can be built up from the flag values or you can use one of the predefined values below. When testing formats always use the FORMAT_FLAG macros to test for individual features - future versions of the library may add new flags.

When *reading* or *writing color-mapped images* the format should be set to the format of the entries in the color-map then png_image_{read,write}_colormap called to read or write the color-map and set the format correctly for the image data. Do not set the PNG_FORMAT_FLAG_COLORMAP bit directly!

NOTE: libpng can be built with particular features disabled. If you see *compiler errors* because the definition of one of the following flags has been compiled out it is because libpng does not have the required support. It is possible, however, for the libpng configuration to enable the format on just read or just write; in that case you may see an *error* at run time. You can guard against this by checking for the definition of the appropriate "_SUPPORTED" macro, one of:

```
   PNG_SIMPLIFIED_{READ,WRITE}_{BGR,AFIRST}_SUPPORTED

   PNG_FORMAT_FLAG_ALPHA    format with an alpha channel
   PNG_FORMAT_FLAG_COLOR    color format: otherwise grayscale
   PNG_FORMAT_FLAG_LINEAR   2-byte channels else 1-byte
   PNG_FORMAT_FLAG_COLORMAP image data is color-mapped
   PNG_FORMAT_FLAG_BGR      BGR colors, else order is RGB
   PNG_FORMAT_FLAG_AFIRST   alpha channel comes first
```

Supported formats are as follows. Future versions of libpng may support more formats; for compatibility with older versions simply check if the format macro is defined using #ifdef. These defines describe the in-memory layout of the components of the pixels of the image.

First the single byte (sRGB) formats:

```
   PNG_FORMAT_GRAY
   PNG_FORMAT_GA
   PNG_FORMAT_AG
   PNG_FORMAT_RGB
   PNG_FORMAT_BGR
   PNG_FORMAT_RGBA
   PNG_FORMAT_ARGB
   PNG_FORMAT_BGRA
   PNG_FORMAT_ABGR
```

Then the linear 2-byte formats. When naming these "Y" is used to indicate a luminance (gray) channel. The component order within the pixel is always the same - there is no provision for swapping the order of the components in the linear format. The components are 16-bit integers in the native byte order for your platform, and there is *no provision* for swapping the bytes to a *different endian condition*.

```
   PNG_FORMAT_LINEAR_Y
   PNG_FORMAT_LINEAR_Y_ALPHA
   PNG_FORMAT_LINEAR_RGB
   PNG_FORMAT_LINEAR_RGB_ALPHA
```

With color-mapped formats the image data is one byte for each pixel. The byte is an index into the color-map which is formatted as above. To obtain a color-mapped format it is sufficient just to add the PNG_FOMAT_FLAG_COLORMAP to one of the above definitions, or you can use one of the definitions below.

```
    PNG_FORMAT_RGB_COLORMAP
    PNG_FORMAT_BGR_COLORMAP
    PNG_FORMAT_RGBA_COLORMAP
    PNG_FORMAT_ARGB_COLORMAP
    PNG_FORMAT_BGRA_COLORMAP
    PNG_FORMAT_ABGR_COLORMAP
```

4. PNG_IMAGE macros

These are convenience macros to derive information from a png_image structure. The PNG_IMAGE_SAMPLE_ macros return values appropriate to the actual image sample values - either the *entries in the color-map* or the *pixels in the image*. The PNG_IMAGE_PIXEL_ macros return *corresponding values for the pixel*s and will always return 1 *for color-mapped formats*. The remaining macros return information about the rows in the image and the complete image.

NOTE: All the macros that take a png_image::format parameter are compile time constants if the format parameter is, itself, a constant. Therefore these macros can be used in array declarations and case labels where required. Similarly the macros are also pre-processor constants (sizeof is not used) so they can be used in #if tests.

```
   PNG_IMAGE_SAMPLE_CHANNELS(fmt)
     Returns the total number of channels in a given format: 1..4

   PNG_IMAGE_SAMPLE_COMPONENT_SIZE(fmt)
     Returns the size in bytes of a single component of a pixel or color-map
     entry (as appropriate) in the image: 1 or 2.

   PNG_IMAGE_SAMPLE_SIZE(fmt)
     This is the size of the sample data for one sample.  If the image is
     color-mapped it is the size of one color-map entry (and image pixels are
     one byte in size), otherwise it is the size of one image pixel.

   PNG_IMAGE_MAXIMUM_COLORMAP_COMPONENTS(fmt)
     The maximum size of the color-map required by the format expressed in a
     count of components.  This can be used to compile-time allocate a
     color-map:

     png_uint_16 colormap[PNG_IMAGE_MAXIMUM_COLORMAP_COMPONENTS(linear_fmt)];

     png_byte colormap[PNG_IMAGE_MAXIMUM_COLORMAP_COMPONENTS(sRGB_fmt)];

     Alternatively use the PNG_IMAGE_COLORMAP_SIZE macro below to use the
     information from one of the png_image_begin_read_ APIs and dynamically
     allocate the required memory.

   PNG_IMAGE_COLORMAP_SIZE(fmt)
    The size of the color-map required by the format; this is the size of the
    color-map buffer passed to the png_image_{read,write}_colormap APIs. It is
    a fixed number determined by the format so can easily be allocated on the
    stack if necessary.

Corresponding information about the pixels
```

```
   PNG_IMAGE_PIXEL_CHANNELS(fmt)
    The number of separate channels (components) in a pixel; 1 for a
    color-mapped image.

   PNG_IMAGE_PIXEL_COMPONENT_SIZE(fmt)\
    The size, in bytes, of each component in a pixel; 1 for a color-mapped
    image.

   PNG_IMAGE_PIXEL_SIZE(fmt)
    The size, in bytes, of a complete pixel; 1 for a color-mapped image.

Information about the whole row, or whole image

   PNG_IMAGE_ROW_STRIDE(image)
    Returns the total number of components in a single row of the image; this
    is the minimum 'row stride', the minimum count of components between each
    row.  For a color-mapped image this is the minimum number of bytes in a
    row.

    If you need the stride measured in bytes, row_stride_bytes is
    PNG_IMAGE_ROW_STRIDE(image) * PNG_IMAGE_PIXEL_COMPONENT_SIZE(fmt)
    plus any padding bytes that your application might need, for example
    to start the next row on a 4-byte boundary.

   PNG_IMAGE_BUFFER_SIZE(image, row_stride)
    Return the size, in bytes, of an image buffer given a png_image and a row
    stride - the number of components to leave space for in each row.

   PNG_IMAGE_SIZE(image)
    Return the size, in bytes, of the image in memory given just a png_image;
    the row stride is the minimum stride required for the image.

   PNG_IMAGE_COLORMAP_SIZE(image)
    Return the size, in bytes, of the color-map of this image.  If the image
    format is not a color-map format this will return a size sufficient for
    256 entries in the given format; check PNG_FORMAT_FLAG_COLORMAP if
    you don't want to allocate a color-map in this case.

PNG_IMAGE_FLAG_*

Flags containing additional information about the image are held in the
'flags' field of png_image.

   PNG_IMAGE_FLAG_COLORSPACE_NOT_sRGB == 0x01
     This indicates that the RGB values of the in-memory bitmap do not
     correspond to the red, green and blue end-points defined by sRGB.

   PNG_IMAGE_FLAG_FAST == 0x02
    On write emphasise speed over compression; the resultant PNG file will be
    larger but will be produced significantly faster, particular for large
    images.  Do not use this option for images which will be distributed, only
    used it when producing intermediate files that will be read back in
    repeatedly.  For a typical 24-bit image the option will double the read
    speed at the cost of increasing the image size by 25%, however for many
    more compressible images the PNG file can be 10 times larger with only a
    slight speed gain.

   PNG_IMAGE_FLAG_16BIT_sRGB == 0x04
     On read if the image is a 16-bit per component image and there is no gAMA
     or sRGB chunk assume that the components are sRGB encoded.  Notice that
     images output by the simplified API always have gamma information; setting
     this flag only affects the interpretation of 16-bit images from an
     external source.  It is recommended that the application expose this flag
     to the user; the user can normally easily recognize the difference between
     linear and sRGB encoding.  This flag has no effect on write - the data
     passed to the write APIs must have the correct encoding (as defined
```

```
    above.)

    If the flag is not set (the default) input 16-bit per component data is
    assumed to be linear.

    NOTE: the flag can only be set after the png_image_begin_read_ call,
    because that call initializes the 'flags' field.

READ APIs

   The png_image passed to the read APIs must have been initialized by setting
   the png_controlp field 'opaque' to NULL (or, better, memset the whole
thing.)

   int png_image_begin_read_from_file( png_imagep image,
     const char *file_name)

      The named file is opened for read and the image header
      is filled in from the PNG header in the file.

   int png_image_begin_read_from_stdio (png_imagep image,
     FILE* file)

       The PNG header is read from the stdio FILE object.

   int png_image_begin_read_from_memory(png_imagep image,
      png_const_voidp memory, size_t size)

      The PNG header is read from the given memory buffer.

   int png_image_finish_read(png_imagep image,
      png_colorp background, void *buffer,
      png_int_32 row_stride, void *colormap));

      Finish reading the image into the supplied buffer and
      clean up the png_image structure.

      row_stride is the step, in png_byte or png_uint_16 units
      as appropriate, between adjacent rows.  A positive stride
      indicates that the top-most row is first in the buffer -
      the normal top-down arrangement.  A negative stride
      indicates that the bottom-most row is first in the buffer.

      background need only be supplied if an alpha channel must
      be removed from a png_byte format and the removal is to be
      done by compositing on a solid color; otherwise it may be
      NULL and any composition will be done directly onto the
      buffer.  The value is an sRGB color to use for the
      background, for grayscale output the green channel is used.

      For linear output removing the alpha channel is always done
      by compositing on black.

   void png_image_free(png_imagep image)

      Free any data allocated by libpng in image->opaque,
      setting the pointer to NULL.  May be called at any time
      after the structure is initialized.
```

When the simplified API needs to convert between sRGB and linear colorspaces, the actual sRGB transfer curve defined in the sRGB specification (see the article at https://en.wikipedia.org/wiki/SRGB) is used, *not the gamma=1/2.2 approximation* used elsewhere in libpng.

5. WRITE APIS

For write you must initialize a `png_image` structure to describe the image to be written:

```
   version: must be set to PNG_IMAGE_VERSION
   opaque: must be initialized to NULL
   width: image width in pixels
   height: image height in rows
   format: the format of the data you wish to write
   flags: set to 0 unless one of the defined flags applies; set
      PNG_IMAGE_FLAG_COLORSPACE_NOT_sRGB for color format images
      where the RGB values do not correspond to the colors in sRGB.
   colormap_entries: set to the number of entries in the color-map (0 to 256)

   int png_image_write_to_file, (png_imagep image,
      const char *file, int convert_to_8bit, const void *buffer,
      png_int_32 row_stride, const void *colormap));

      Write the image to the named file.

   int png_image_write_to_memory (png_imagep image, void *memory,
      png_alloc_size_t * PNG_RESTRICT memory_bytes,
      int convert_to_8_bit, const void *buffer, ptrdiff_t row_stride,
      const void *colormap));

      Write the image to memory.

   int png_image_write_to_stdio(png_imagep image, FILE *file,
      int convert_to_8_bit, const void *buffer,
      png_int_32 row_stride, const void *colormap)

      Write the image to the given (FILE*).
```

With all write APIs if image is in one of the linear formats with (`png_uint_16`) data then setting `convert_to_8_bit` will cause the output to be a (`png_byte`) PNG gamma encoded according to the sRGB specification, otherwise a 16-bit linear encoded PNG file is written.

With all APIs `row_stride` is handled as in the read APIs - it is the spacing from one row to the next in component sized units (`float`) and if *negative* indicates a bottom-up row layout in the buffer. If you pass *zero*, libpng will calculate the `row_stride` for you from the width and number of channels.

Note that the write API does not support interlacing, sub-8-bit pixels, indexed (paletted) images, or most ancillary chunks.

VI. 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. Both of those are compile-time issues; that is, they are generally determined at the time the code is written, and there is rarely a need to provide the user with a means of changing them.

1. Memory allocation, input/output, and error handling

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_malloc()`, `png_calloc()`, and `png_free()`. The `png_malloc()` and `png_free()` functions currently just call the standard C functions and `png_calloc()` calls `png_malloc()` and then clears the newly allocated memory to *zero*; note that `png_calloc(png_ptr, size)` is not the same as the `calloc(number, size)` function provided by `stdlib.h`. There is limited support for certain systems with segmented memory architectures and the *types of pointers* declared by `png.h` match this; you will have to use appropriate pointers in your application. If you prefer to use a different method of allocating and freeing data, you can use `png_create_read_struct_2()` or `png_create_write_struct_2()` to register your own functions as described above. These functions also provide a `void pointer` that can be retrieved via

```
    mem_ptr=png_get_mem_ptr(png_ptr);
```

Your replacement memory functions must have prototypes as follows:

```
    png_voidp malloc_fn(png_structp png_ptr,
       png_alloc_size_t size);

    void free_fn(png_structp png_ptr, png_voidp ptr);
```

Your `malloc_fn()` must return `NULL` in case of failure. The `png_malloc()` function will normally call `png_error()` if it receives a `NULL` from the system memory allocator or from your replacement `malloc_fn()`.

Your `free_fn()` will never be called with a `NULL` `ptr`, since libpng's `png_free()` checks for `NULL` before calling `free_fn()`.

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 must have prototypes as follows:

    void user_read_data(png_structp png_ptr,
```

```
        png_bytep data, size_t length);

    void user_write_data(png_structp png_ptr,
        png_bytep data, size_t length);

    void user_flush_data(png_structp png_ptr);
```

The user_read_data() function is responsible for detecting and handling *end-of-data errors*.

Supplying NULL for the *read*, *write*, or *flush* functions sets them back to using the default C stream functions, which expect the io_ptr to point to a standard *FILE structure. It is probably a mistake to use NULL for one of write_data_fn and output_flush_fn but not both of them, unless you have built libpng with PNG_NO_WRITE_FLUSH defined. 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_NO_SETJMP, in which case it is handled via PNG_ABORT()), but you could change this to do things like exit() if you should wish, *as long as your function does not return*.

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_CONSOLE_IO defined (because you don't want the *messages*) or PNG_NO_STDIO defined (because fprintf() isn't available). 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 redirect *errors* and *warnings* to your own replacement 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);
```

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);
```

Then, within your user_error_fn or user_warning_fn, you can retrieve the error_ptr if you need it, by calling

```
    png_voidp error_ptr = png_get_error_ptr(png_ptr);
```

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. For an alternative approach, you may wish to use the "cexcept" facility (see https://cexcept.sourceforge.io/), which is illustrated in pngvalid.c and in contrib/visupng.

Beginning in libpng-1.4.0, the png_set_benign_errors() API became available. You can use this to handle certain *errors* (normally handled as *errors*) as *warnings*.

```
    png_set_benign_errors (png_ptr, int allowed);

    allowed: 0: treat png_benign_error() as an error.
             1: treat png_benign_error() as a warning.
```

As of libpng-1.6.0, the default condition is to treat benign *errors* as *warnings* while reading and as *errors* while writing.

2. 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. However, 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 private or unknown chunks in a generic method, via *callback functions*, instead of by modifying libpng functions. This is illustrated in pngtest.c, which uses a *callback function* to handle a private "**vpAg**" chunk and the new "**sTER**" chunk, which are both unknown to libpng.

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.

3. 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.).

4. 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:

```
    #include zlib.h
    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). Note that the memory level does have an effect on compression; among other things, lower levels will result in sections of incompressible data being emitted in smaller stored blocks, with a correspondingly larger relative overhead of up to 15% in the worst case.

```
    #include zlib.h
    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.

```
    #include zlib.h
    png_set_compression_strategy(png_ptr,
        strategy);

    png_set_compression_window_bits(png_ptr,
        window_bits);
```

```
    png_set_compression_method(png_ptr, method);
```

This controls the size of the **`IDAT`** chunks (default 8192):

```
    png_set_compression_buffer_size(png_ptr, size);
```

As of libpng version 1.5.4, additional APIs became available to set these separately for non-**`IDAT`** compressed chunks such as **`zTXt`**, **`iTXt`**, and **`iCCP`**:

```
    #include zlib.h
    #if PNG_LIBPNG_VER >= 10504
    png_set_text_compression_level(png_ptr, level);

    png_set_text_compression_mem_level(png_ptr, level);

    png_set_text_compression_strategy(png_ptr,
        strategy);

    png_set_text_compression_window_bits(png_ptr,
        window_bits);
```

```
    png_set_text_compression_method(png_ptr, method);
    #endif
```

5. 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`, `PNG_NO_FILTERS`, or `PNG_FAST_FILTERS` to turn filtering `on` and `off`, or to turn `on` just the fast-decoding subset of filters, respectively.

Individual filter types are `PNG_FILTER_NONE`, `PNG_FILTER_SUB`, `PNG_FILTER_UP`, `PNG_FILTER_AVG`, `PNG_FILTER_PAETH`, which can be bitwise `OR`ed together with '|' 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. (Note that this means the first row must always be adaptively filtered, because libpng currently does not allocate the filter buffers until `png_write_row()` is called for the first time.)

```
    filters = PNG_NO_FILTERS;
    filters = PNG_ALL_FILTERS;
    filters = PNG_FAST_FILTERS;

    or

    filters = PNG_FILTER_NONE | PNG_FILTER_SUB |
              PNG_FILTER_UP | PNG_FILTER_AVG |
              PNG_FILTER_PAETH;

    png_set_filter(png_ptr, PNG_FILTER_TYPE_BASE,
       filters);

              The second parameter can also be
              PNG_INTRAPIXEL_DIFFERENCING if you are
              writing a PNG to be embedded in a MNG
              datastream.  This parameter must be the
              same as the value of filter_method used
              in png_set_IHDR().
```

6. 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", 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.

VII. MNG support

The MNG specification (available at http://www.libpng.org/pub/mng) allows certain extensions to PNG for PNG images that are embedded in MNG datastreams. Libpng can support some of these extensions. To enable them, use the png_permit_mng_features() function:

```
   feature_set = png_permit_mng_features(png_ptr, mask)

   mask is a png_uint_32 containing the bitwise OR of the
        features you want to enable.  These include
        PNG_FLAG_MNG_EMPTY_PLTE
        PNG_FLAG_MNG_FILTER_64
        PNG_ALL_MNG_FEATURES

   feature_set is a png_uint_32 that is the bitwise AND of
      your mask with the set of MNG features that is
      supported by the version of libpng that you are using.
```

It is an *error* to use this function when reading or writing a standalone PNG file with the PNG 8-byte signature. The PNG datastream must be wrapped in a MNG datastream. As a minimum, it must have the MNG 8-byte signature and the **MHDR** and **MEND** chunks. Libpng does not provide support for these or any

other MNG chunks; your application must provide its own support for them. You may wish to consider using libmng (available at https://www.libmng.com/) instead.

VIII. 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_destroy()` have been moved to `PNG_INTERNAL` in version 0.95 to discourage their use. These functions will be removed from libpng version 1.4.0.

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.

Support for the **`sCAL`**, **`iCCP`**, **`iTXt`**, and **`sPLT`** chunks was added at libpng-1.0.6; however, **`iTXt`** support was not enabled by default.

Starting with version 1.0.7, you can find out which version of the library you are using at run-time:

```
   png_uint_32 libpng_vn = png_access_version_number();
```

The number `libpng_vn` is constructed from the major version, minor version with leading *zero*, and release number with leading *zero*, (e.g., `libpng_vn` for version `1.0.7` is `10007`).

Note that this function does not take a `png_ptr`, so you can call it before you've created one.

You can also check which version of `png.h` you used when compiling your application:

```
   png_uint_32 application_vn = PNG_LIBPNG_VER;
```

IX. Changes to Libpng from version 1.0.x to 1.2.x

Support for user memory management was enabled by default. To accomplish this, the functions `png_create_read_struct_2()`, `png_create_write_struct_2()`, `png_set_mem_fn()`, `png_get_mem_ptr()`, `png_malloc_default()`, and `png_free_default()` were added.

Support for the **`iTXt`** chunk has been enabled by default as of version 1.2.41.

Support for certain MNG features was enabled.

Support for numbered *error messages* was added. However, we never got around to actually numbering the *error messages*. The function `png_set_strip_error_numbers()` was added (Note: the prototype for this function was inadvertently removed from `png.h` in `PNG_NO_ASSEMBLER_CODE` builds of libpng-1.2.15. It was restored in libpng-1.2.36).

The `png_malloc_warn()` function was added at libpng-1.2.3. This issues a `png_warning` and returns `NULL` instead of aborting when it fails to acquire the requested memory allocation.

Support for setting user limits on image width and height was enabled by default. The functions `png_set_user_limits()`, `png_get_user_width_max()`, and `png_get_user_height_max()` were added at libpng-1.2.6.

The `png_set_add_alpha()` function was added at libpng-1.2.7.

The function `png_set_expand_gray_1_2_4_to_8()` was added at libpng-1.2.9. Unlike `png_set_gray_1_2_4_to_8()`, the new function does not expand the **`tRNS`** chunk to alpha. The `png_set_gray_1_2_4_to_8()` function is deprecated.

A number of macro definitions in support of runtime selection of assembler code features (especially Intel MMX code support) were added at libpng-1.2.0:

```
PNG_ASM_FLAG_MMX_SUPPORT_COMPILED
PNG_ASM_FLAG_MMX_SUPPORT_IN_CPU
PNG_ASM_FLAG_MMX_READ_COMBINE_ROW
PNG_ASM_FLAG_MMX_READ_INTERLACE
PNG_ASM_FLAG_MMX_READ_FILTER_SUB
PNG_ASM_FLAG_MMX_READ_FILTER_UP
PNG_ASM_FLAG_MMX_READ_FILTER_AVG
PNG_ASM_FLAG_MMX_READ_FILTER_PAETH
PNG_ASM_FLAGS_INITIALIZED
PNG_MMX_READ_FLAGS
PNG_MMX_FLAGS
PNG_MMX_WRITE_FLAGS
PNG_MMX_FLAGS
```

We added the following functions in support of runtime selection of assembler code features:

```
png_get_mmx_flagmask()
png_set_mmx_thresholds()
png_get_asm_flags()
png_get_mmx_bitdepth_threshold()
png_get_mmx_rowbytes_threshold()
png_set_asm_flags()
```

We replaced all of these functions with simple stubs in libpng-1.2.20, when the Intel assembler code was removed due to a licensing issue.

These macros are deprecated:

```
PNG_READ_TRANSFORMS_NOT_SUPPORTED
PNG_PROGRESSIVE_READ_NOT_SUPPORTED
PNG_NO_SEQUENTIAL_READ_SUPPORTED
PNG_WRITE_TRANSFORMS_NOT_SUPPORTED
PNG_READ_ANCILLARY_CHUNKS_NOT_SUPPORTED
PNG_WRITE_ANCILLARY_CHUNKS_NOT_SUPPORTED
```

They have been replaced, respectively, by:

```
PNG_NO_READ_TRANSFORMS
PNG_NO_PROGRESSIVE_READ
PNG_NO_SEQUENTIAL_READ
PNG_NO_WRITE_TRANSFORMS
PNG_NO_READ_ANCILLARY_CHUNKS
PNG_NO_WRITE_ANCILLARY_CHUNKS
```

`PNG_MAX_UINT` was replaced with `PNG_UINT_31_MAX`. It has been deprecated since libpng-1.0.16 and libpng-1.2.6.

The function

```
png_check_sig(sig, num)
```

was replaced with

```
!png_sig_cmp(sig, 0, num)
```

It has been deprecated since libpng-0.90.

The function

```
png_set_gray_1_2_4_to_8()
```

which also expands **`tRNS`** to alpha was replaced with

```
png_set_expand_gray_1_2_4_to_8()
```

which does not. It has been deprecated since libpng-1.0.18 and 1.2.9.

X. Changes to Libpng from version 1.0.x/1.2.x to 1.4.x

Private libpng prototypes and macro definitions were moved from `png.h` and `pngconf.h` into a new `pngpriv.h` header file.

Functions `png_set_benign_errors()`, `png_benign_error()`, and `png_chunk_benign_error()` were added.

Support for setting the maximum amount of memory that the application will allocate for reading chunks was added, as a security measure. The functions `png_set_chunk_cache_max()` and `png_get_chunk_cache_max()` were added to the library.

We implemented support for I/O states by adding png_ptr member io_state and functions png_get_io_chunk_name() and png_get_io_state() in pngget.c

We added PNG_TRANSFORM_GRAY_TO_RGB to the available high-level input transforms.

Checking for and reporting of *errors* in the **IHDR** chunk is more thorough.

Support for global arrays was removed, to improve thread safety.

Some obsolete/deprecated macros and functions have been removed.

Typecasted NULL definitions such as

```
   #define png_voidp_NULL           (png_voidp)NULL
```

were eliminated. If you used these in your application, just use NULL instead.

The png_struct and info_struct members "trans" and "trans_values" were changed to "trans_alpha" and "trans_color", respectively.

The obsolete, unused pnggccrd.c and pngvcrd.c files and related makefiles were removed.

The PNG_1_0_X and PNG_1_2_X macros were eliminated.

The PNG_LEGACY_SUPPORTED macro was eliminated.

Many WIN32_WCE #ifdefs were removed.

The functions png_read_init(info_ptr), png_write_init(info_ptr), png_info_init(info_ptr), png_read_destroy(), and png_write_destroy() have been removed. They have been deprecated since libpng-0.95.

The png_permit_empty_plte() was removed. It has been deprecated since libpng-1.0.9. Use png_permit_mng_features() instead.

We removed the obsolete stub functions png_get_mmx_flagmask(), png_set_mmx_thresholds(), png_get_asm_flags(), png_get_mmx_bitdepth_threshold(), png_get_mmx_rowbytes_threshold(), png_set_asm_flags(), and png_mmx_supported()

We removed the obsolete png_check_sig(), png_memcpy_check(), and png_memset_check() functions. Instead use !png_sig_cmp(), memcpy(), and memset(), respectively.

The function png_set_gray_1_2_4_to_8() was removed. It has been deprecated since libpng-1.0.18 and 1.2.9, when it was replaced with png_set_expand_gray_1_2_4_to_8() because the former function also expanded any **tRNS** chunk to an alpha channel.

Macros for png_get_uint_16, png_get_uint_32, and png_get_int_32 were added and are used by default instead of the corresponding functions. Unfortunately, from libpng-1.4.0 until 1.4.4, the png_get_uint_16 macro (but not the function) incorrectly returned a *value of type* png_uint_32.

We changed the prototype for png_malloc() from

```
    png_malloc(png_structp png_ptr, png_uint_32 size)
```

to

```
    png_malloc(png_structp png_ptr, png_alloc_size_t size)
```

This also applies to the prototype for the user replacement `malloc_fn()`.

The `png_calloc()` function was added and is used in place of of "`png_malloc(); memset();`" except in the case in `png_read_png()` where the *array consists of pointers*; in this case a "`for`" loop is used after the `png_malloc()` to set the pointers to `NULL`, to give robust. behavior in case the application runs out of memory part-way through the process.

We changed the prototypes of `png_get_compression_buffer_size()` and `png_set_compression_buffer_size()` to work with `size_t` instead of `png_uint_32`.

Support for numbered *error messages* was removed by default, since we never got around to actually numbering the *error messages*. The function `png_set_strip_error_numbers()` was removed from the library by default.

The `png_zalloc()` and `png_zfree()` functions are no longer exported. The `png_zalloc()` function *no longer zeroes* out the memory that it allocates. Applications that called `png_zalloc(png_ptr, number, size)` can call `png_calloc(png_ptr, number*size)` instead, and can call `png_free()` instead of `png_zfree()`.

Support for dithering was disabled by default in libpng-1.4.0, because it has not been well tested and doesn't actually "dither". The code was not removed, however, and could be enabled by building libpng with `PNG_READ_DITHER_SUPPORTED` defined. In libpng-1.4.2, this support was re-enabled, but the function was renamed `png_set_quantize()` to reflect more accurately what it actually does. At the same time, the `PNG_DITHER_[RED,GREEN_BLUE]_BITS` macros were also renamed to `PNG_QUANTIZE_[RED,GREEN,BLUE]_BITS`, and `PNG_READ_DITHER_SUPPORTED` was renamed to `PNG_READ_QUANTIZE_SUPPORTED`.

We removed the trailing '.' from the *warning* and *error messages*.

XI. Changes to Libpng from version 1.4.x to 1.5.x

From libpng-1.4.0 until 1.4.4, the `png_get_uint_16` macro (but not the function) incorrectly returned a *value of type* `png_uint_32`. The incorrect macro was removed from libpng-1.4.5.

Checking for invalid palette index on write was added at libpng 1.5.10. If a pixel contains an invalid (out-of-range) index libpng issues a *benign error*. This is enabled by default because this condition is an *error* according to the PNG specification, Clause 11.3.2, but the *error* can be ignored in each `png_ptr` with

```
   png_set_check_for_invalid_index(png_ptr, allowed);

      allowed  - one of
                 0: disable benign error (accept the
                    invalid data without warning).
```

```
                1: enable benign error (treat the
                   invalid data as an error or a
                   warning).
```

If the *error* is ignored, or if png_benign_error() treats it as a *warning*, any *invalid pixels* are decoded as *opaque black* by the decoder and written as-is by the encoder.

Retrieving the maximum palette index found was added at libpng-1.5.15. This statement must appear after png_read_png() or png_read_image() while reading, and after png_write_png() or png_write_image() while writing.

```
   int max_palette = png_get_palette_max(png_ptr, info_ptr);
```

This will return the *maximum palette index* found in the image, or "-1" if the palette was not checked, or "0" if no palette was found. Note that this does not account for any palette index used by ancillary chunks such as the **bKGD** chunk; you must check those separately to determine the maximum palette index actually used.

There are no substantial API changes between the non-deprecated parts of the 1.4.5 API and the 1.5.0 API; however, the ability to directly access members of the main libpng control structures, png_struct and png_info, deprecated in earlier versions of libpng, has been completely removed from libpng 1.5, and new private "pngstruct.h", "pnginfo.h", and "pngdebug.h" header files were created.

We no longer include zlib.h in png.h. The include statement has been moved to pngstruct.h, where it is not accessible by applications. Applications that need access to information in zlib.h will need to add the '#include "zlib.h"' directive. It does not matter whether this is placed prior to or after the '"#include png.h"' directive.

The png_sprintf(), png_strcpy(), and png_strncpy() macros are no longer used and were removed.

We moved the png_strlen(), png_memcpy(), png_memset(), and png_memcmp() macros into a private header file (pngpriv.h) that is not accessible to applications.

In png_get_iCCP, the type of "profile" was changed from png_charpp to png_bytepp, and in png_set_iCCP, from png_charp to png_const_bytep.

There are changes of form in png.h, including new and changed macros to declare parts of the API. Some API functions with arguments that are pointers to data not modified within the function have been corrected to declare these arguments with const.

Much of the internal use of C macros to control the library build has also changed and some of this is visible in the exported header files, in particular the use of macros to control data and API elements visible during application compilation may require significant revision to application code. (It is extremely rare for an application to do this.)

Any program that compiled against libpng 1.4 and did not use deprecated features or access internal library structures should compile and work against libpng 1.5, except for the change in the prototype for png_get_iCCP() and png_set_iCCP() API functions mentioned above.

libpng 1.5.0 adds PNG_ PASS macros to help in the reading and writing of interlaced images. The macros return the number of rows and columns in each pass and information that can be used to de-interlace and (if absolutely necessary) interlace an image.

libpng 1.5.0 adds an API png_longjmp(png_ptr, value). This API calls the application-provided png_longjmp_ptr on the internal, but application initialized, longjmp buffer. It is provided as a convenience to avoid the need to use the png_jmpbuf macro, which had the unnecessary side effect of resetting the internal png_longjmp_ptr value.

libpng 1.5.0 includes a complete fixed point API. By default this is present along with the corresponding floating point API. In general the fixed point API is faster and smaller than the floating point one because the PNG file format used fixed point, not floating point. This applies even if the library uses floating point in internal calculations. A new macro, PNG_FLOATING_ARITHMETIC_SUPPORTED, reveals whether the library uses floating point arithmetic (the default) or fixed point arithmetic internally for performance critical calculations such as gamma correction. In some cases, the gamma calculations may produce slightly different results. This has changed the results in png_rgb_to_gray and in alpha composition (png_set_background for example). This applies even if the original image was already linear (gamma == 1.0) and, therefore, it is not necessary to linearize the image. This is because libpng has **not** been changed to optimize that case correctly, yet.

Fixed point support for the **sCAL** chunk comes with an important caveat; the **sCAL** specification uses a decimal encoding of floating point values and the accuracy of PNG fixed point values is insufficient for representation of these values. Consequently a "string" API (png_get_sCAL_s and png_set_sCAL_s) is the only reliable way of reading arbitrary **sCAL** chunks in the absence of either the floating point API or internal floating point calculations. Starting with libpng-1.5.0, both of these functions are present when PNG_sCAL_SUPPORTED is defined. Prior to libpng-1.5.0, their presence also depended upon PNG_FIXED_POINT_SUPPORTED being defined and PNG_FLOATING_POINT_SUPPORTED not being defined.

Applications no longer need to include the optional distribution header file pngusr.h or define the corresponding macros during application build in order to see the correct variant of the libpng API. From 1.5.0 application code can check for the corresponding _SUPPORTED macro:

```
#ifdef PNG_INCH_CONVERSIONS_SUPPORTED
   /* code that uses the inch conversion APIs. */
#endif
```

This macro will only be defined if the inch conversion functions have been compiled into libpng. The full set of macros, and whether or not support has been compiled in, are available in the header file pnglibconf.h. This header file is specific to the libpng build. Notice that prior to 1.5.0 the _SUPPORTED macros would always have the default definition unless reset by pngusr.h or by explicit settings on the compiler command line. These settings may produce compiler *warnings* or *errors* in 1.5.0 because of macro redefinition.

Applications can now choose whether to use these macros or to call the corresponding function by defining PNG_USE_READ_MACROS or PNG_NO_USE_READ_MACROS before including png.h. Notice that this is only supported from 1.5.0; defining PNG_NO_USE_READ_MACROS prior to 1.5.0 will lead to a link failure. Prior to libpng-1.5.4, the zlib compressor used the same set of parameters when compressing the **IDAT** data and textual data such as **zTXt** and **iCCP**. In libpng-1.5.4 we reinitialized the zlib stream for each type of data. We added five png_set_text_*() functions for setting the parameters to use with textual data.

Prior to libpng-1.5.4, the PNG_READ_16_TO_8_ACCURATE_SCALE_SUPPORTED option was off by default, and slightly inaccurate scaling occurred. This option can no longer be turned off, and the choice of accurate or inaccurate 16-to-8 scaling is by using the new png_set_scale_16_to_8() API for accurate scaling or the old png_set_strip_16_to_8() API for simple chopping. In libpng-1.5.4, the PNG_READ_16_TO_8_ACCURATE_SCALE_SUPPORTED macro became PNG_READ_SCALE_16_TO_8_SUPPORTED, and the PNG_READ_16_TO_8 macro became PNG_READ_STRIP_16_TO_8_SUPPORTED, to enable the two png_set_*_16_to_8() functions separately.

Prior to libpng-1.5.4, the png_set_user_limits() function could only be used to reduce the width and height limits from the value of PNG_USER_WIDTH_MAX and PNG_USER_HEIGHT_MAX, although this document said that it could be used to override them. Now this function will reduce or increase the limits.

Starting in libpng-1.5.22, default user limits were established. These can be overridden by application calls to png_set_user_limits(), png_set_user_chunk_cache_max(), and/or png_set_user_malloc_max().
The limits are now

```
                          max possible  default
   png_user_width_max        0x7fffffff    1,000,000
   png_user_height_max       0x7fffffff    1,000,000
   png_user_chunk_cache_max  0 (unlimited) 1000
   png_user_chunk_malloc_max 0 (unlimited) 8,000,000
```

The png_set_option() function (and the "options" member of the png struct) was added to libpng-1.5.15, with option PNG_ARM_NEON.

The library now supports a complete fixed point implementation and can thus be used on systems that have no floating point support or very limited or slow support. Previously gamma correction, an essential part of complete PNG support, required reasonably fast floating point.

As part of this the choice of internal implementation has been made independent of the choice of fixed versus floating point APIs and all the missing fixed point APIs have been implemented.

The exact mechanism used to control attributes of API functions has changed, as described in the INSTALL file.

A new test program, pngvalid, is provided in addition to pngtest. pngvalid validates the arithmetic accuracy of the gamma correction calculations and includes a number of validations of the file format. A subset of the full range of tests is run when "make check" is done (in the 'configure' build.) pngvalid also allows total allocated memory usage to be evaluated and performs additional memory overwrite validation.

Many changes to individual feature macros have been made. The following are the changes most likely to be noticed by library builders who configure libpng:

1) All feature macros now have consistent naming:

```
#define PNG_NO_feature turns the feature off
#define PNG_feature_SUPPORTED turns the feature on
```

pnglibconf.h contains one line for each feature macro which is either:

```
#define PNG_feature_SUPPORTED
```

if the feature is supported or:

```
/*#undef PNG_feature_SUPPORTED*/
```

if it is not. Library code consistently checks for the 'SUPPORTED' macro. It does not, and libpng applications should not, check for the 'NO' macro which will not normally be defined even if the feature is not supported. The 'NO' macros are only used internally for setting or not setting the corresponding 'SUPPORTED' macros.

Compatibility with the old names is provided as follows:

PNG_INCH_CONVERSIONS turns on PNG_INCH_CONVERSIONS_SUPPORTED

And the following definitions disable the corresponding feature:

PNG_SETJMP_NOT_SUPPORTED disables SETJMP
PNG_READ_TRANSFORMS_NOT_SUPPORTED disables READ_TRANSFORMS
PNG_NO_READ_COMPOSITED_NODIV disables READ_COMPOSITE_NODIV
PNG_WRITE_TRANSFORMS_NOT_SUPPORTED disables WRITE_TRANSFORMS
PNG_READ_ANCILLARY_CHUNKS_NOT_SUPPORTED disables READ_ANCILLARY_CHUNKS
PNG_WRITE_ANCILLARY_CHUNKS_NOT_SUPPORTED disables WRITE_ANCILLARY_CHUNKS

Library builders should remove use of the above, inconsistent, names.

2) *Warning* and *error message* formatting was previously conditional on the STDIO feature. The library has been changed to use the CONSOLE_IO feature instead. This means that if CONSOLE_IO is disabled the library no longer uses the printf(3) functions, even though the default read/write implementations use (FILE) style stdio.h functions.

3) Three feature macros now control the fixed/floating point decisions:

PNG_FLOATING_POINT_SUPPORTED enables the floating point APIs

PNG_FIXED_POINT_SUPPORTED enables the fixed point APIs; however, in practice these are normally required internally anyway (because the PNG file format is fixed point), therefore in most cases PNG_NO_FIXED_POINT merely stops the function from being exported.

PNG_FLOATING_ARITHMETIC_SUPPORTED chooses between the internal floating point implementation or the fixed point one. Typically the fixed point implementation is larger and slower than the floating point implementation on a system that supports floating point; however, it may be faster on a system which lacks floating point hardware and therefore uses a software emulation.

4) Added PNG_{READ,WRITE}_INT_FUNCTIONS_SUPPORTED. This allows the functions to read and write ints to be disabled independently of PNG_USE_READ_MACROS, which allows libpng to be built with the functions even though the default is to use the macros - this allows applications to choose at app buildtime whether or not to use macros (previously impossible because the functions weren't in the default build.)

XII. Changes to Libpng from version 1.5.x to 1.6.x

A "*simplified API*" has been added (see documentation in `png.h` and a simple example in `contrib/examples/pngtopng.c`). The new publicly visible API includes the following:

```
   macros:
     PNG_FORMAT_*
     PNG_IMAGE_*
   structures:
     png_control
     png_image
   read functions
     png_image_begin_read_from_file()
     png_image_begin_read_from_stdio()
     png_image_begin_read_from_memory()
     png_image_finish_read()
     png_image_free()
   write functions
     png_image_write_to_file()
     png_image_write_to_memory()
     png_image_write_to_stdio()
```

Starting with libpng-1.6.0, you can configure libpng to prefix all exported symbols, using the `PNG_PREFIX` macro.

We no longer include `string.h` in `png.h`. The include statement has been moved to `pngpriv.h`, where it is not accessible by applications. Applications that need access to information in `string.h` must add an '`#include <string.h>`' directive. It does not matter whether this is placed prior to or after the '`#include "png.h"`' directive.

The following API are now DEPRECATED:

```
   png_info_init_3()
   png_convert_to_rfc1123() which has been replaced
     with png_convert_to_rfc1123_buffer()
   png_malloc_default()
   png_free_default()
   png_reset_zstream()
```

The following have been removed:

```
   png_get_io_chunk_name(), which has been replaced
     with png_get_io_chunk_type().  The new
     function returns a 32-bit integer instead of
     a string.
   The png_sizeof(), png_strlen(), png_memcpy(), png_memcmp(), and
     png_memset() macros are no longer used in the libpng sources and
     have been removed.  These had already been made invisible to applications
     (i.e., defined in the private pngpriv.h header file) since libpng-1.5.0.
```

The signatures of many exported functions were changed, such that

```
   png_structp became png_structrp or png_const_structrp
   png_infop became png_inforp or png_const_inforp
```

where "`rp`" indicates a "*restricted pointer*".

Dropped support for 16-bit platforms. The support for FAR/far types has been eliminated and the definition of png_alloc_size_t is now controlled by a flag so that 'small size_t' systems can select it if necessary.

Error detection in some chunks has improved; in particular the **iCCP** chunk reader now does pretty complete validation of the basic format. Some bad profiles that were previously accepted are now accepted with a *warning* or *rejected*, depending upon the png_set_benign_errors() setting, in particular the very old broken Microsoft/HP 3144-byte sRGB profile. Starting with libpng-1.6.11, recognizing and checking sRGB profiles can be avoided by means of

```
#if defined(PNG_SKIP_sRGB_CHECK_PROFILE) && \
    defined(PNG_SET_OPTION_SUPPORTED)
   png_set_option(png_ptr, PNG_SKIP_sRGB_CHECK_PROFILE,
       PNG_OPTION_ON);
#endif
```

It's not a good idea to do this if you are using the "*simplified API*", which needs to be able to recognize sRGB profiles conveyed via the **iCCP** chunk.

The PNG spec requirement that only grayscale profiles may appear in images with color type 0 or 4 and that even if the image only contains gray pixels, only RGB profiles may appear in images with color type 2, 3, or 6, is now enforced. The sRGB chunk is allowed to appear in images with any color type and is interpreted by libpng to convey a one-tracer-curve gray profile or a three-tracer-curve RGB profile as appropriate.

Libpng 1.5.x erroneously used /MD for Debug DLL builds; if you used the debug builds in your app and you changed your app to use /MD you will need to change it back to /MDd for libpng 1.6.x.

Prior to libpng-1.6.0 a *warning* would be issued if the **iTXt** chunk contained an empty language field or an empty translated keyword. Both of these are allowed by the PNG specification, so these *warnings* are no longer issued.

The library now issues an *error* if the application attempts to set a transform after it calls png_read_update_info() or if it attempts to call both png_read_update_info() and png_start_read_image() or to call either of them more than once.

The default condition for benign_errors is now to treat *benign errors* as *warnings* while reading and as *errors* while writing.

The library now issues a *warning* if both background processing and RGB to gray are used when gamma correction happens. As with previous versions of the library the results are numerically very incorrect in this case.

There are some minor arithmetic changes in some transforms such as png_set_background(), that might be detected by certain regression tests.

Unknown chunk handling has been improved internally, without any API change. This adds more correct option control of the unknown handling, corrects a pre-existing bug where the per-chunk 'keep' setting is ignored, and makes it possible to skip IDAT chunks in the sequential reader.

The machine-generated configure files are no longer included in branches libpng16 and later of the GIT repository. They continue to be included in the tarball releases, however.

Libpng-1.6.0 through 1.6.2 used the *CMF* bytes at the beginning of the IDAT stream to set the size of the sliding window for reading instead of using the default 32-kbyte sliding window size. It was discovered that there are hundreds of PNG files in the wild that have incorrect *CMF* bytes that caused zlib to issue the "invalid distance too far back" *error* and *reject* the file. Libpng-1.6.3 and later calculate their own safe *CMF* from the image dimensions, provide a way to revert to the libpng-1.5.x behavior (ignoring the *CMF* bytes and using a 32-kbyte sliding window), by using

```
    png_set_option(png_ptr, PNG_MAXIMUM_INFLATE_WINDOW,
        PNG_OPTION_ON);
```

and provide a tool (`contrib/tools/pngfix`) for rewriting a PNG file while optimizing the *CMF* bytes in its **IDAT** chunk correctly.

Libpng-1.6.0 and libpng-1.6.1 wrote uncompressed **iTXt** chunks with the wrong length, which resulted in PNG files that cannot be read beyond the bad **iTXt** chunk. This error was fixed in libpng-1.6.3, and a tool (called `contrib/tools/png-fix-itxt`) has been added to the libpng distribution.

Starting with libpng-1.6.17, the PNG_SAFE_LIMITS macro was eliminated and safe limits are used by default (users who need larger limits can still override them at compile time or run time, as described above).

The new limits are

```
                            default   spec limit
   png_user_width_max         1,000,000  2,147,483,647
   png_user_height_max        1,000,000  2,147,483,647
   png_user_chunk_cache_max         128  unlimited
   png_user_chunk_malloc_max  8,000,000  unlimited
```

Starting with libpng-1.6.18, a PNG_RELEASE_BUILD macro was added, which allows library builders to control compilation for an installed system (a release build). It can be set for testing debug or beta builds to ensure that they will compile when the build type is switched to RC or STABLE. In essence this overrides the PNG_LIBPNG_BUILD_BASE_TYPE definition which is not directly user controllable.

Starting with libpng-1.6.19, attempting to set an over-length **PLTE** chunk is an error. Previously this requirement of the PNG specification was not enforced, and the palette was always limited to 256 entries. An over-length **PLTE** chunk found in an input PNG is silently truncated.

Starting with libpng-1.6.31, the **eXIf** chunk is supported. Libpng does not attempt to decode the Exif profile; it simply returns a byte array containing the profile to the calling application which must do its own decoding.

XIII. Detecting libpng

The png_get_io_ptr() function has been present since libpng-0.88, has never changed, and is unaffected by conditional compilation macros. It is the best choice for use in configure scripts for detecting the presence of any libpng version since 0.88. In an `autoconf` "`configure.in`" you could use

```
    AC_CHECK_LIB(png, png_get_io_ptr, ...
```

XIV. Source code repository

Since about February 2009, version 1.2.34, libpng has been under "git" source control. The git repository was built from old libpng-x.y.z.tar.gz files going back to version 0.70. You can access the git repository (read only) at

https://github.com/glennrp/libpng or
https://git.code.sf.net/p/libpng/code.git

or you can browse it with a web browser at

https://github.com/glennrp/libpng or
https://sourceforge.net/p/libpng/code/ci/libpng16/tree/

Patches can be sent to png-mng-implement at lists.sourceforge.net or uploaded to the libpng bug tracker at

https://libpng.sourceforge.io/

or as a "pull request" to

https://github.com/glennrp/libpng/pulls

We also accept patches built from the tar or zip distributions, and simple verbal descriptions of bug fixes, reported either to the SourceForge bug tracker, to the `png-mng-implement` at `lists.sf.net` mailing list, as github issues.

XV. Coding style

Our coding style is similar to the "*Allman*" style (See https://en.wikipedia.org/wiki/Indent_style#Allman_style), with curly braces on separate lines:

```
    if (condition)
    {
       action;
    }

    else if (another condition)
    {
       another action;
    }

The braces can be omitted from simple one-line actions:

    if (condition)
```

```
      return 0;
```

We use 3-space indentation, except for continued statements which are usually indented the same as the first line of the statement plus four more spaces.

For macro definitions we use 2-space indentation, always leaving the "#" in the first column.

```
   #ifndef PNG_NO_FEATURE
   #  ifndef PNG_FEATURE_SUPPORTED
   #    define PNG_FEATURE_SUPPORTED
   #  endif
   #endif
```

Comments appear with the leading "/*" at the same indentation as the statement that follows the comment:

```
   /* Single-line comment */
   statement;

   /* This is a multiple-line
    * comment.
    */
   statement;
```

Very short comments can be placed after the end of the statement to which they pertain:

```
   statement;    /* comment */
```

We don't use C++ style ("//") comments. We have, however, used them in the past in some now-abandoned MMX assembler code.

Functions and their curly braces are not indented, and exported functions are marked with PNGAPI:

```
/* This is a public function that is visible to
 * application programmers. It does thus-and-so.
 */
void PNGAPI
png_exported_function(png_ptr, png_info, foo)
{
   body;
}
```

The return type and decorations are placed on a separate line ahead of the function name, as illustrated above.

The prototypes for all exported functions appear in png.h, above the comment that says

```
   /* Maintainer: Put new public prototypes here ... */
```

We mark all non-exported functions with "/* PRIVATE */"":

```
void /* PRIVATE */
png_non_exported_function(png_ptr, png_info, foo)
{
   body;
}
```

The prototypes for non-exported functions (except for those in `pngtest`) appear in `pngpriv.h` above the comment that says

```
  /* Maintainer: Put new private prototypes here ^ */
```

To avoid polluting the global namespace, the names of all exported functions and variables begin with "`png_`", and all publicly visible C preprocessor macros begin with "`PNG`". We request that applications that use libpng **not** begin any of their own symbols with either of these strings.

We put a space after the "`sizeof`" operator and we omit the optional parentheses around its argument when the argument is an expression, not a type name, and we always enclose the `sizeof` operator, with its argument, in parentheses:

```
  (sizeof (png_uint_32))
  (sizeof array)
```

Prior to libpng-1.6.0 we used a "`png_sizeof()`" macro, formatted as though it were a function.

Control keywords `if`, `for`, `while`, and `switch` are always followed by a space to distinguish them from function calls, which have no trailing space.

We put a space after each comma and after each semicolon in "`for`" statements, and we put spaces before and after each C binary operator and after "`for`" or "`while`", and before "`?`". We don't put a space between a *typecast* and the *expression being cast*, nor do we put one between a *function name* and the *left parenthesis* that follows it:

```
   for (i = 2; i > 0; --i)
      y[i] = a(x) + (int)b;
```

We prefer `#ifdef` and `#ifndef` to `#if defined()` and `#if !defined()` when there is only one macro being tested. We always use parentheses with "`defined`".

We express integer constants that are used as bit masks in hex format, with an even number of lower-case hex digits, and to make them unsigned (e.g., `0x00U`, `0xffU`, `0x0100U`) and long if they are greater than `0x7fff` (e.g., `0xffffUL`).

We prefer to use underscores rather than *camelCase* in names, except for a few type names that we inherit from `zlib.h`.

We prefer "`if (something != 0)`" and "`if (something == 0)`" over "`if (something)`" and "`if (!something)`", respectively, and for pointers we prefer "`if (some_pointer != NULL)`" or "`if (some_pointer == NULL)`".

We do not use the `TAB` character for indentation in the C sources.

Lines do not exceed 80 characters.

Other rules can be inferred by inspecting the libpng source.