NAME

libpng − Portable Network Graphics (PNG) Reference Library 1.2.8

SYNOPSIS

#include <png.h>

png_uint_32 png_access_version_number *(void*)**;**

int png_check_sig (png_bytep *sig*, **int** *num*)**;**

void png_chunk_error (png_structp *png_ptr*, **png_const_charp** *error*)**;**

void png_chunk_warning (png_structp *png_ptr*, **png_const_charp** *message*)**;**

void png_convert_from_struct_tm (png_timep *ptime*, **struct tm FAR *** *ttime*)**;**

void png_convert_from_time_t (png_timep *ptime*, **time_t** *ttime*)**;**

png_charp png_convert_to_rfc1123 (png_structp *png_ptr*, **png_timep** *ptime*)**;**

png_infop png_create_info_struct (png_structp *png_ptr*)**;**

png_structp png_create_read_struct (png_const_charp *user_png_ver*, **png_voidp** *error_ptr*, **png_error_ptr** *error_fn*, **png_error_ptr** *warn_fn*)**;**

png_structp png_create_read_struct_2(png_const_charp *user_png_ver*, **png_voidp** *error_ptr*, **png_error_ptr** *error_fn*, **png_error_ptr** *warn_fn*, **png_voidp** *mem_ptr*, **png_malloc_ptr** *malloc_fn*, **png_free_ptr** *free_fn*)**;**

png_structp png_create_write_struct (png_const_charp *user_png_ver*, **png_voidp** *error_ptr*, **png_error_ptr** *error_fn*, **png_error_ptr** *warn_fn*)**;**

png_structp png_create_write_struct_2(png_const_charp *user_png_ver*, **png_voidp** *error_ptr*, **png_error_ptr** *error_fn*, **png_error_ptr** *warn_fn*, **png_voidp** *mem_ptr*, **png_malloc_ptr** *malloc_fn*, **png_free_ptr** *free_fn*);

int png_debug(int *level*, **png_const_charp** *message*);

int png_debug1(int *level*, **png_const_charp** *message*, *p1*);

int png_debug2(int *level*, **png_const_charp** *message*, *p1*, *p2*);

void png_destroy_info_struct (png_structp *png_ptr*, **png_infopp** *info_ptr_ptr*);

void png_destroy_read_struct (png_structpp *png_ptr_ptr*, **png_infopp** *info_ptr_ptr*, **png_infopp** *end_info_ptr_ptr*);

void png_destroy_write_struct (png_structpp *png_ptr_ptr*, **png_infopp** *info_ptr_ptr*);

void png_error (png_structp *png_ptr*, **png_const_charp** *error*);

void png_free (png_structp *png_ptr*, **png_voidp** *ptr*);

void png_free_chunk_list (png_structp *png_ptr*);

void png_free_default(png_structp *png_ptr*, **png_voidp** *ptr*);

void png_free_data (png_structp *png_ptr*, **png_infop** *info_ptr*, **int** *num*);

png_byte png_get_bit_depth (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_uint_32 png_get_bKGD (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_color_16p** **back-ground*);

png_byte png_get_channels (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_uint_32 png_get_cHRM (png_structp *png_ptr*, **png_infop** *info_ptr*, **double** **white_x*, **double** **white_y*, **double** **red_x*, **double** **red_y*, **double** **green_x*, **double** **green_y*, **double** **blue_x*, **double** **blue_y*);

png_uint_32 png_get_cHRM_fixed (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** **white_x*, **png_uint_32** **white_y*, **png_uint_32** **red_x*, **png_uint_32** **red_y*, **png_uint_32** **green_x*, **png_uint_32** **green_y*, **png_uint_32** **blue_x*, **png_uint_32** **blue_y*);

png_byte png_get_color_type (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_byte png_get_compression_type (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_byte png_get_copyright (png_structp *png_ptr*);

png_voidp png_get_error_ptr (png_structp *png_ptr*);

png_byte png_get_filter_type (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_uint_32 png_get_gAMA (png_structp *png_ptr*, **png_infop** *info_ptr*, **double** **file_gamma*);

png_uint_32 png_get_gAMA_fixed (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** **int_file_gamma*);

png_byte png_get_header_ver (png_structp *png_ptr*);

png_byte png_get_header_version (png_structp *png_ptr*);

png_uint_32 png_get_hIST (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_16p** **hist*);

png_uint_32 png_get_iCCP (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_charpp** *name*, **int** **compression_type*, **png_charpp** *profile*, **png_uint_32** **proflen*);

png_uint_32 png_get_IHDR (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** **width*, **png_uint_32** **height*, **int** **bit_depth*, **int** **color_type*, **int** **interlace_type*, **int** **compression_type*, **int** **filter_type*);

png_uint_32 png_get_image_height (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_uint_32 png_get_image_width (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_byte png_get_interlace_type (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_voidp png_get_io_ptr (png_structp *png_ptr*);

png_byte png_get_libpng_ver (png_structp *png_ptr*);

png_voidp png_get_mem_ptr(png_structp *png_ptr*);

png_uint_32 png_get_oFFs (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** **offset_x*, **png_uint_32** **offset_y*, **int** **unit_type*);

png_uint_32 png_get_pCAL (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_charp** **purpose*, **png_int_32** **X0*, **png_int_32** **X1*, **int** **type*, **int** **nparams*, **png_charp** **units*, **png_charpp** **params*);

png_uint_32 png_get_pHYs (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** **res_x*, **png_uint_32** **res_y*, **int** **unit_type*);

float png_get_pixel_aspect_ratio (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_uint_32 png_get_pixels_per_meter (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_voidp png_get_progressive_ptr (png_structp *png_ptr*);

png_uint_32 png_get_PLTE (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_colorp** **palette*, **int** **num_palette*);

png_byte png_get_rgb_to_gray_status (png_structp *png_ptr)*

png_uint_32 png_get_rowbytes (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_bytepp png_get_rows (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_uint_32 png_get_sBIT (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_color_8p** **sig_bit*);

png_bytep png_get_signature (png_structp *png_ptr*, **png_infop** *info_ptr*);

png_uint_32 png_get_sPLT (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_spalette_p** **splt_ptr*);

png_uint_32 png_get_sRGB (png_structp *png_ptr*, **png_infop** *info_ptr*, **int** **intent*);

png_uint_32 png_get_text (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_textp** **text_ptr*, **int** **num_text*);

png_uint_32 png_get_tIME (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_timep** **mod_time*);

png_uint_32 png_get_tRNS (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_bytep** **trans*, **int** **num_trans*, **png_color_16p** **trans_values*);

png_uint_32 png_get_unknown_chunks (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_unknown_chunkpp** *unknowns*);

png_voidp png_get_user_chunk_ptr (png_structp *png_ptr*);

png_uint_32 png_get_user_height_max(png_structp *png_ptr*);

png_voidp png_get_user_transform_ptr (png_structp *png_ptr*);

png_uint_32 png_get_user_width_max (png_structp *png_ptr***);**

png_uint_32 png_get_valid (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *flag***);**

png_int_32 png_get_x_offset_microns (png_structp *png_ptr***, png_infop** *info_ptr***);**

png_int_32 png_get_x_offset_pixels (png_structp *png_ptr***, png_infop** *info_ptr***);**

png_uint_32 png_get_x_pixels_per_meter (png_structp *png_ptr***, png_infop** *info_ptr***);**

png_int_32 png_get_y_offset_microns (png_structp *png_ptr***, png_infop** *info_ptr***);**

png_int_32 png_get_y_offset_pixels (png_structp *png_ptr***, png_infop** *info_ptr***);**

png_uint_32 png_get_y_pixels_per_meter (png_structp *png_ptr***, png_infop** *info_ptr***);**

png_uint_32 png_get_compression_buffer_size (png_structp *png_ptr***);**

int png_handle_as_unknown (png_structp *png_ptr***, png_bytep** *chunk_name***);**

void png_init_io (png_structp *png_ptr***, FILE** **fp***);*

DEPRECATED: void png_info_init (png_infop *info_ptr***);**

DEPRECATED: void png_info_init_2 (png_infopp *ptr_ptr***, png_size_t** *png_info_struct_size***);**

png_voidp png_malloc (png_structp *png_ptr***, png_uint_32** *size***);**

png_voidp png_malloc_default(png_structp *png_ptr***, png_uint_32** *size***);**

voidp png_memcpy (png_voidp *s1*, **png_voidp** *s2*, **png_size_t** *size*);

png_voidp png_memcpy_check (png_structp *png_ptr*, **png_voidp** *s1*, **png_voidp** *s2*, **png_uint_32** *size*);

voidp png_memset (png_voidp *s1*, **int** *value*, **png_size_t** *size*);

png_voidp png_memset_check (png_structp *png_ptr*, **png_voidp** *s1*, **int** *value*, **png_uint_32** *size*);

DEPRECATED: void png_permit_empty_plte (png_structp *png_ptr*, **int** *empty_plte_permitted*);

void png_process_data (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_bytep** *buffer*, **png_size_t** *buffer_size*);

void png_progressive_combine_row (png_structp *png_ptr*, **png_bytep** *old_row*, **png_bytep** *new_row*);

void png_read_destroy (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_infop** *end_info_ptr*);

void png_read_end (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_read_image (png_structp *png_ptr*, **png_bytepp** *image*);

DEPRECATED: void png_read_init (png_structp *png_ptr*);

DEPRECATED: void png_read_init_2 (png_structpp *ptr_ptr*, **png_const_charp** *user_png_ver*, **png_size_t** *png_struct_size*, **png_size_t** *png_info_size*);

void png_read_info (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_read_png (png_structp *png_ptr*, **png_infop** *info_ptr*, **int** *transforms*, **png_voidp** *params*);

void png_read_row (png_structp *png_ptr*, **png_bytep** *row*, **png_bytep** *display_row*);

void png_read_rows (png_structp *png_ptr*, **png_bytepp** *row*, **png_bytepp** *display_row*, **png_uint_32** *num_rows*);

void png_read_update_info (png_structp *png_ptr*, **png_infop** *info_ptr*);

#if *!defined(PNG_1_0_X)*

void png_set_add_alpha (png_structp *png_ptr*, **png_uint_32** *filler*, **int** *flags*);

#endif

void png_set_background (png_structp *png_ptr*, **png_color_16p** *background_color*, **int** *background_gamma_code*, **int** *need_expand*, **double** *background_gamma*);

void png_set_bgr (png_structp *png_ptr*);

void png_set_bKGD (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_color_16p** *background*);

void png_set_cHRM (png_structp *png_ptr*, **png_infop** *info_ptr*, **double** *white_x*, **double** *white_y*, **double** *red_x*, **double** *red_y*, **double** *green_x*, **double** *green_y*, **double** *blue_x*, **double** *blue_y*);

void png_set_cHRM_fixed (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *white_x*, **png_uint_32** *white_y*, **png_uint_32** *red_x*, **png_uint_32** *red_y*, **png_uint_32** *green_x*, **png_uint_32** *green_y*, **png_uint_32** *blue_x*, **png_uint_32** *blue_y*);

void png_set_compression_level (png_structp *png_ptr*, **int** *level*);

void png_set_compression_mem_level (png_structp *png_ptr*, **int** *mem_level*);

void png_set_compression_method (png_structp *png_ptr*, **int** *method*);

void png_set_compression_strategy (png_structp *png_ptr***, int** *strategy***);**

void png_set_compression_window_bits (png_structp *png_ptr***, int** *window_bits***);**

void png_set_crc_action (png_structp *png_ptr***, int** *crit_action***, int** *ancil_action***);**

void png_set_dither (png_structp *png_ptr***, png_colorp** *palette***, int** *num_palette***, int** *maximum_colors***, png_uint_16p** *histogram***, int** *full_dither***);**

void png_set_error_fn (png_structp *png_ptr***, png_voidp** *error_ptr***, png_error_ptr** *error_fn***, png_error_ptr** *warning_fn***);**

void png_set_expand (png_structp *png_ptr***);**

void png_set_filler (png_structp *png_ptr***, png_uint_32** *filler***, int** *flags***);**

void png_set_filter (png_structp *png_ptr***, int** *method***, int** *filters***);**

void png_set_filter_heuristics (png_structp *png_ptr***, int** *heuristic_method***, int** *num_weights***, png_doublep** *filter_weights***, png_doublep** *filter_costs***);**

void png_set_flush (png_structp *png_ptr***, int** *nrows***);**

void png_set_gamma (png_structp *png_ptr***, double** *screen_gamma***, double** *default_file_gamma***);**

void png_set_gAMA (png_structp *png_ptr***, png_infop** *info_ptr***, double** *file_gamma***);**

void png_set_gAMA_fixed (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *file_gamma***);**

void png_set_gray_1_2_4_to_8(png_structp *png_ptr***);**

void png_set_gray_to_rgb (png_structp *png_ptr***);**

void png_set_hIST (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_16p** *hist***);**

void png_set_iCCP (png_structp *png_ptr***, png_infop** *info_ptr***, png_charp** *name***, int** *compression_type***, png_charp** *profile***, png_uint_32** *proflen***);**

int png_set_interlace_handling (png_structp *png_ptr***);**

void png_set_invalid (png_structp *png_ptr***, png_infop** *info_ptr***, int** *mask***);**

void png_set_invert_alpha (png_structp *png_ptr***);**

void png_set_invert_mono (png_structp *png_ptr***);**

void png_set_IHDR (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *width***, png_uint_32** *height***, int** *bit_depth***, int** *color_type***, int** *interlace_type***, int** *compression_type***, int** *filter_type***);**

void png_set_keep_unknown_chunks (png_structp *png_ptr***, int** *keep***, png_bytep** *chunk_list***, int** *num_chunks***);**

void png_set_mem_fn(png_structp *png_ptr***, png_voidp** *mem_ptr***, png_malloc_ptr** *malloc_fn***, png_free_ptr** *free_fn***);**

void png_set_oFFs (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *offset_x***, png_uint_32** *offset_y***, int** *unit_type***);**

void png_set_packing (png_structp *png_ptr***);**

void png_set_packswap (png_structp *png_ptr***);**

void png_set_palette_to_rgb(png_structp *png_ptr***);**

void png_set_pCAL (png_structp *png_ptr***, png_infop** *info_ptr***, png_charp** *purpose***, png_int_32** *X0***, png_int_32** *X1***, int** *type***, int** *nparams***, png_charp** *units***, png_charpp** *params***);**

void png_set_pHYs (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *res_x***, png_uint_32** *res_y***, int** *unit_type***);**

void png_set_progressive_read_fn (png_structp *png_ptr***, png_voidp** *progressive_ptr***, png_progressive_info_ptr** *info_fn***, png_progressive_row_ptr** *row_fn***, png_progressive_end_ptr** *end_fn***);**

void png_set_PLTE (png_structp *png_ptr***, png_infop** *info_ptr***, png_colorp** *palette***, int** *num_palette***);**

void png_set_read_fn (png_structp *png_ptr***, png_voidp** *io_ptr***, png_rw_ptr** *read_data_fn***);**

void png_set_read_status_fn (png_structp *png_ptr***, png_read_status_ptr** *read_row_fn***);**

void png_set_read_user_transform_fn (png_structp *png_ptr***, png_user_transform_ptr** *read_user_transform_fn***);**

void png_set_rgb_to_gray (png_structp *png_ptr***, int** *error_action***, double** *red***, double** *green***);**

void png_set_rgb_to_gray_fixed (png_structp *png_ptr***, int error_action png_fixed_point** *red***, png_fixed_point** *green***);**

void png_set_rows (png_structp *png_ptr***, png_infop** *info_ptr***, png_bytepp** *row_pointers***);**

void png_set_sBIT (png_structp *png_ptr***, png_infop** *info_ptr***, png_color_8p** *sig_bit***);**

void png_set_sCAL (png_structp *png_ptr***, png_infop** *info_ptr***, png_charp** *unit***, double** *width***, double** *height***);**

void png_set_shift (png_structp *png_ptr***, png_color_8p** *true_bits***);**

void png_set_sig_bytes (png_structp *png_ptr***, int** *num_bytes***);**

void png_set_sPLT (png_structp *png_ptr***, png_infop** *info_ptr***, png_spalette_p** *splt_ptr***, int** *num_spalettes***);**

void png_set_sRGB (png_structp *png_ptr***, png_infop** *info_ptr***, int** *intent***);**

void png_set_sRGB_gAMA_and_cHRM (png_structp *png_ptr***, png_infop** *info_ptr***, int** *intent***);**

void png_set_strip_16 (png_structp *png_ptr***);**

void png_set_strip_alpha (png_structp *png_ptr***);**

void png_set_swap (png_structp *png_ptr***);**

void png_set_swap_alpha (png_structp *png_ptr***);**

void png_set_text (png_structp *png_ptr***, png_infop** *info_ptr***, png_textp** *text_ptr***, int** *num_text***);**

void png_set_tIME (png_structp *png_ptr***, png_infop** *info_ptr***, png_timep** *mod_time***);**

void png_set_tRNS (png_structp *png_ptr***, png_infop** *info_ptr***, png_bytep** *trans***, int** *num_trans***, png_color_16p** *trans_values***);**

void png_set_tRNS_to_alpha(png_structp *png_ptr***);**

png_uint_32 png_set_unknown_chunks (png_structp *png_ptr***, png_infop** *info_ptr***, png_unknown_chunkp** *unknowns***, int** *num***, int** *location***);**

void png_set_unknown_chunk_location(png_structp *png_ptr***, png_infop** *info_ptr***, int** *chunk***, int** *location***);**

void png_set_read_user_chunk_fn (png_structp *png_ptr***, png_voidp** *user_chunk_ptr***, png_user_chunk_ptr** *read_user_chunk_fn***);**

void png_set_user_limits (png_structp *png_ptr***, png_uint_32** *user_width_max***, png_uint_32** *user_height_max***);**

void png_set_user_transform_info (png_structp *png_ptr***, png_voidp** *user_transform_ptr***, int** *user_transform_depth***, int** *user_transform_channels***);**

void png_set_write_fn (png_structp *png_ptr***, png_voidp** *io_ptr***, png_rw_ptr** *write_data_fn***, png_flush_ptr** *output_flush_fn***);**

void png_set_write_status_fn (png_structp *png_ptr***, png_write_status_ptr** *write_row_fn***);**

void png_set_write_user_transform_fn (png_structp *png_ptr***, png_user_transform_ptr** *write_user_transform_fn***);**

void png_set_compression_buffer_size(png_structp *png_ptr***, png_uint_32** *size***);**

int png_sig_cmp (png_bytep *sig***, png_size_t** *start***, png_size_t** *num_to_check***);**

void png_start_read_image (png_structp *png_ptr***);**

void png_warning (png_structp *png_ptr***, png_const_charp** *message***);**

void png_write_chunk (png_structp *png_ptr***, png_bytep** *chunk_name***, png_bytep** *data***, png_size_t** *length***);**

void png_write_chunk_data (png_structp *png_ptr***, png_bytep** *data***, png_size_t** *length***);**

void png_write_chunk_end (png_structp *png_ptr***);**

void png_write_chunk_start (png_structp *png_ptr*, **png_bytep** *chunk_name*, **png_uint_32** *length*);

void png_write_destroy (png_structp *png_ptr*);

void png_write_end (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_write_flush (png_structp *png_ptr*);

void png_write_image (png_structp *png_ptr*, **png_bytepp** *image*);

DEPRECATED: void png_write_init (png_structp *png_ptr*);

DEPRECATED: void png_write_init_2 (png_structpp *ptr_ptr*, **png_const_charp** *user_png_ver*, **png_size_t** *png_struct_size*, **png_size_t** *png_info_size*);

void png_write_info (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_write_info_before_PLTE (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_write_png (png_structp *png_ptr*, **png_infop** *info_ptr*, **int** *transforms*, **png_voidp** *params*);

void png_write_row (png_structp *png_ptr*, **png_bytep** *row*);

void png_write_rows (png_structp *png_ptr*, **png_bytepp** *row*, **png_uint_32** *num_rows*);

voidpf png_zalloc (voidpf *png_ptr*, **uInt** *items*, **uInt** *size*);

void png_zfree (voidpf *png_ptr*, **voidpf** *ptr*);

DESCRIPTION

The *libpng* library supports encoding, decoding, and various manipulations of the Portable Network Graphics (PNG) format image files. It uses the *zlib(3)* compression library. Following is a copy of the libpng.txt file that accompanies libpng.

LIBPNG.TXT

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

libpng version 1.2.8 - December 3, 2004
Updated and distributed by Glenn Randers-Pehrson
<glennrp at users.sourceforge.net>

based on:

libpng 1.0 beta 6 version 0.96 May 28, 1997
Updated and distributed by Andreas Dilger

libpng 1.0 beta 2 - version 0.88 January 26, 1996

Updated/rewritten per request in the libpng FAQ

December 18, 1995 & January 20, 1996

I. Introduction

This file describes how to use and modify the PNG reference library (known as libpng) for your own use. There are five sections to this file: introduction, structures, reading, writing, and modification and configuration notes for various special platforms. In addition to this file, example.c is a good starting point for using the library, as it is heavily commented and should include everything most people will need. We assume that libpng is already installed; see the INSTALL file for instructions on how to install libpng.

Libpng was written as a companion to the PNG specification, as a way of reducing the amount of time and effort it takes to support the PNG file format in application programs.

The PNG specification (second edition), November 2003, is available as a W3C Recommendation and as an ISO Standard (ISO/IEC 15948:2003 (E)) at <http://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 <http://www.libpng.org/pub/png/documents/>

The PNG-1.0 specification is available as RFC 2083 <http://www.libpng.org/pub/png/documents/> and as a W3C Recommendation <http://www.w3.org/TR/REC.png.html>. Some additional chunks are described in the special-purpose public chunks documents at <http://www.libpng.org/pub/png/documents/>.

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, <http://www.info-zip.org/pub/infozip/zlib/>. The zlib compression utility is a general purpose utility that is useful for more than PNG files, and can be used without libpng. See the documentation delivered with zlib for more details. You can usually find the source files for the zlib utility wherever you find the libpng source files.

Libpng is thread safe, provided the threads are using different instances of the structures. Each thread should have its own png_struct and png_info instances, and thus its own image. Libpng does not protect itself against two threads using the same instance of a structure. Note: thread safety may be defeated by use of some of the MMX assembler code in pnggccrd.c, which is only compiled when the user defines PNG_THREAD_UNSAFE_OK.

II. Structures

There are two main structures that are important to libpng, png_struct and png_info. The first, png_struct, is an internal structure that will not, for the most part, be used by a user except as the first variable passed to every libpng function call.

The png_info structure is designed to provide information about the PNG file. At one time, the fields of png_info were intended to be directly accessible to the user. However, this tended to cause problems with applications using dynamically loaded libraries, and as a result a set of interface functions for png_info (the png_get_*() and png_set_*() functions) was developed. The fields of png_info are still available for older applications, but it is suggested that applications use the new interfaces if at all possible.

Applications that do make direct access to the members of png_struct (except for png_ptr->jmpbuf) must be recompiled whenever the library is updated, and applications that make direct access to the members of png_info must be recompiled if they were compiled or loaded with libpng version 1.0.6, in which the members were in a different order. In version 1.0.7, the members of the png_info structure reverted to the old order, as they were in versions 0.97c through 1.0.5. Starting with version 2.0.0, both structures are going to be hidden, and the contents of the structures will only be accessible through the png_get/png_set functions.

The png.h header file is an invaluable reference for programming with libpng. And while I'm on the topic, make sure you include the libpng header file:

#include <png.h>

III. Reading

We'll now walk you through the possible functions to call when reading in a PNG file sequentially, briefly explaining the syntax and purpose of each one. See example.c and png.h for more detail. While progressive reading is covered in the next section, you will still need some of the functions discussed in this section to read a PNG file.

Setup

You will want to do the I/O initialization(*) before you get into libpng, so if it doesn't work, you don't have much to undo. Of course, you will also want to insure that you are, in fact, dealing with a PNG file. Libpng provides a simple check to see if a file is a PNG file. To use it, pass in the first 1 to 8 bytes of the file to the function png_sig_cmp(), and it will return 0 if the bytes match the corresponding bytes of the PNG signature, or nonzero 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_read() with the number of bytes you read from the beginning. Libpng will then only check the bytes (if any) that your program didn't read.

(*): If you are not using the standard I/O functions, you will need to replace them with custom functions. See the discussion under Customizing libpng.

```
FILE *fp = fopen(file_name, "rb");
if (!fp)
{
   return (ERROR);
}
fread(header, 1, number, fp);
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);
}

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);
}
```

If you want to use your own memory allocation routines, define PNG_USER_MEM_SUPPORTED and use png_create_read_struct_2() instead of png_create_read_struct():

```
png_structp png_ptr = png_create_read_struct_2
   (PNG_LIBPNG_VER_STRING, (png_voidp)user_error_ptr,
    user_error_fn, user_warning_fn, (png_voidp)
    user_mem_ptr, user_malloc_fn, user_free_fn);
```

The error handling routines passed to png_create_read_struct() and the memory alloc/free routines passed to png_create_struct_2() are only necessary if you are not using the libpng supplied error handling and memory alloc/free functions.

When libpng encounters an error, it expects to longjmp back to your routine. Therefore, you will need to call setjmp and pass your png_jmpbuf(png_ptr). If you read the file from different routines, you will need to update the jmpbuf field every time you enter a new routine that will call a png_*() function.

See your documentation of setjmp/longjmp for your compiler for more information on setjmp/longjmp. See the discussion on libpng error handling in the Customizing Libpng section below for more information on the libpng error handling. If an error occurs, and libpng longjmp's back to your setjmp, you will want to call png_destroy_read_struct() to free any memory.

```
if (setjmp(png_jmpbuf(png_ptr)))
{
   png_destroy_read_struct(&png_ptr, &info_ptr,
     &end_info);
   fclose(fp);
   return (ERROR);
}
```

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

Now you need to set up the input code. The default for libpng is to use the C function fread(). If you use this, you will need to pass a valid FILE * in the function png_init_io(). Be sure that the file is opened in binary mode. If you wish to handle reading data in another way, you need not call the png_init_io() function, but you must then implement the libpng I/O methods discussed in the Customizing Libpng section below.

```
png_init_io(png_ptr, fp);
```

If you had previously opened the file and read any of the signature from the beginning in order to see if this was a PNG file, you need to let libpng know that there are some bytes missing from the start of the file.

```
png_set_sig_bytes(png_ptr, number);
```

Setting up callback code

You can set up a callback function to handle any unknown chunks in the input stream. You must supply the function

```
read_chunk_callback(png_ptr ptr,
    png_unknown_chunkp chunk);
{
  /* The unknown chunk structure contains your
     chunk data: */
     png_byte name[5];
     png_byte *data;
     png_size_t size;
  /* Note that libpng has already taken care of
     the CRC handling */

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

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

```
  return (0); /* did not recognize */
  return (n); /* success */
}
```

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

To inform libpng about your function, use

```
png_set_read_user_chunk_fn(png_ptr, user_chunk_ptr,
   read_chunk_callback);
```

This names not only the callback function, but also a user pointer that you can retrieve with

```
png_get_user_chunk_ptr(png_ptr);
```

At this point, you can set up a callback function that will be called after each row has been read, which you can use to control a progress meter or the like. It's demonstrated in pngtest.c. You must supply a function

```
void read_row_callback(png_ptr ptr, png_uint_32 row,
  int pass);
{
 /* put your code here */
}
```

(You can give it another name that you like instead of "read_row_callback")

To inform libpng about your function, use

```
png_set_read_status_fn(png_ptr, read_row_callback);
```

Width and height 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. Since very few applications really need to process such large images, we have imposed an arbitrary 1-million limit on rows and columns. Larger images will be rejected immediately with a png_error() call. If you wish to override this limit, you can use

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

to set your own limits, or use width_max = height_max = 0x7fffffffL to allow all valid dimensions (libpng may reject some very large 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(). 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);
```

Unknown-chunk handling

Now you get to set the way the library processes unknown chunks in the input PNG stream. Both known and unknown chunks will be read. Normal behavior is that known chunks will be parsed into information in various info_ptr members; unknown chunks will be discarded. To change this, you can call:

```
png_set_keep_unknown_chunks(png_ptr, keep,
   chunk_list, num_chunks);
keep       - 0: do not handle as unknown
           1: 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 ' ' if
num_chunks is 0)
num_chunks - number of chunks affected; if 0, all
unknown chunks are affected. If nonzero,
only the chunks in the list 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.

The high-level read interface

At this point there are two ways to proceed; through the high-level read interface, or through a sequence of low-level read operations. You can use the high-level interface if (a) you are willing to read the entire image into memory, and (b) the input transformations you want to do are limited to the following set:

PNG_TRANSFORM_IDENTITY No transformation
PNG_TRANSFORM_STRIP_16 Strip 16-bit samples to
8 bits
PNG_TRANSFORM_STRIP_ALPHA Discard the alpha channel
PNG_TRANSFORM_PACKING Expand 1, 2 and 4-bit
samples to bytes
PNG_TRANSFORM_PACKSWAP Change order of packed
pixels to LSB first
PNG_TRANSFORM_EXPAND Perform set_expand()
PNG_TRANSFORM_INVERT_MONO Invert monochrome images
PNG_TRANSFORM_SHIFT Normalize pixels to the
sBIT depth
PNG_TRANSFORM_BGR Flip RGB to BGR, RGBA
to BGRA
PNG_TRANSFORM_SWAP_ALPHA Flip RGBA to ARGB or GA
to AG
PNG_TRANSFORM_INVERT_ALPHA Change alpha from opacity
to transparency
PNG_TRANSFORM_SWAP_ENDIAN Byte-swap 16-bit samples

(This excludes setting a background color, doing gamma transformation, dithering, and setting filler.) If this is the case, simply do this:

png_read_png(png_ptr, info_ptr, png_transforms, NULL)

where png_transforms is an integer containing the logical OR of some set of transformation flags. This call is equivalent to png_read_info(), followed the set of transformations indicated by the transform mask, 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/png_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*png_sizeof(png_bytep));
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.

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 when you call png_destroy_*().

The low-level read interface

If you are going the low-level route, you are now ready to read all the file information up to the actual image data. You do this with a call to png_read_info().

```
png_read_info(png_ptr, info_ptr);
```

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

Querying the info structure

Functions are used to get the information from the info_ptr once it has been read. Note that these fields may not be completely filled in until png_read_end() has read the chunk data following the image.

```
png_get_IHDR(png_ptr, info_ptr, &width, &height,
  &bit_depth, &color_type, &interlace_type,
  &compression_type, &filter_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

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)
compression_type - (must be PNG_COMPRESSION_TYPE_BASE
                for PNG 1.0)
interlace_type - (PNG_INTERLACE_NONE or
                PNG_INTERLACE_ADAM7)
Any or all of interlace_type, compression_type, of
filter_method can be NULL if you are
not interested in their values.

channels = png_get_channels(png_ptr, info_ptr);
channels       - number of channels of info for the
                color type (valid values are 1 (GRAY,
                PALETTE), 2 (GRAY_ALPHA), 3 (RGB),
                4 (RGB_ALPHA or RGB + filler byte))
rowbytes = png_get_rowbytes(png_ptr, info_ptr);
rowbytes       - number of bytes needed to hold a row

signature = png_get_signature(png_ptr, info_ptr);
signature      - holds the signature read from the
                file (if any).  The data is kept in
                the same offset it would be if the
                whole signature were read (i.e. if an
                application had already read in 4
                bytes of signature before starting
                libpng, the remaining 4 bytes would
                be in signature[4] through signature[7]
                (see png_set_sig_bytes())).


width          = png_get_image_width(png_ptr,
                 info_ptr);
height         = png_get_image_height(png_ptr,
                 info_ptr);
bit_depth       = png_get_bit_depth(png_ptr,
                 info_ptr);
```

```
color_type       = png_get_color_type(png_ptr,
                   info_ptr);
filter_method    = png_get_filter_type(png_ptr,
                   info_ptr);
compression_type = png_get_compression_type(png_ptr,
                   info_ptr);
interlace_type   = png_get_interlace_type(png_ptr,
                   info_ptr);
```

These are also important, but their validity depends on whether the chunk has been read. The png_get_valid(png_ptr, info_ptr, PNG_INFO_<chunk>) and png_get_<chunk>(png_ptr, info_ptr, ...) functions return non-zero if the data has been read, or zero if it is missing. The parameters to the png_get_<chunk> are set directly if they are simple data types, or a pointer into the info_ptr is returned for any complex types.

```
png_get_PLTE(png_ptr, info_ptr, &palette,
             &num_palette);
palette        - the palette for the file
               (array of png_color)
num_palette    - number of entries in the palette

png_get_gAMA(png_ptr, info_ptr, &gamma);
gamma          - the gamma the file is written
               at (PNG_INFO_gAMA)

png_get_sRGB(png_ptr, info_ptr, &srgb_intent);
srgb_intent    - the rendering intent (PNG_INFO_sRGB)
               The presence of the sRGB chunk
               means that the pixel data is in the
               sRGB color space.  This chunk also
               implies specific values of gAMA and
               cHRM.

png_get_iCCP(png_ptr, info_ptr, &name,
  &compression_type, &profile, &proflen);
name           - The profile name.
compression    - The compression type; always
                 PNG_COMPRESSION_TYPE_BASE for PNG 1.0.
                 You may give NULL to this argument to
                 ignore it.
profile        - International Color Consortium color
                 profile data. May contain NULs.
proflen        - length of profile data in bytes.

png_get_sBIT(png_ptr, info_ptr, &sig_bit);
sig_bit        - the number of significant bits for
               (PNG_INFO_sBIT) each of the gray,
               red, green, and blue channels,
               whichever are appropriate for the
               given color type (png_color_16)

png_get_tRNS(png_ptr, info_ptr, &trans, &num_trans,
             &trans_values);
trans          - array of transparent entries for
               palette (PNG_INFO_tRNS)
trans_values   - graylevel or color sample values of
               the single transparent color for
               non-paletted images (PNG_INFO_tRNS)
```

```
num_trans      - number of transparent entries
               (PNG_INFO_tRNS)

png_get_hIST(png_ptr, info_ptr, &hist);
               (PNG_INFO_hIST)
hist           - histogram of palette (array of
               png_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 (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).
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);
palette_ptr    - array of palette structures holding
               contents of one or more sPLT chunks
               read.
num_spalettes  - number of sPLT chunks read.

png_get_oFFs(png_ptr, info_ptr, &offset_x, &offset_y,
  &unit_type);
offset_x       - positive offset from the left edge
               of the screen
```

```
offset_y       - positive offset from the top edge
                 of the screen
unit_type      - PNG_OFFSET_PIXEL, PNG_OFFSET_MICROMETER

png_get_pHYs(png_ptr, info_ptr, &res_x, &res_y,
  &unit_type);
res_x          - pixels/unit physical resolution in
                 x direction
res_y          - pixels/unit physical resolution in
                 x direction
unit_type      - PNG_RESOLUTION_UNKNOWN,
                 PNG_RESOLUTION_METER

png_get_sCAL(png_ptr, info_ptr, &unit, &width,
  &height)
unit       - physical scale units (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
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 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)
```

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)
```

For more information, see the png_info definition in png.h and the PNG specification for chunk contents. Be careful with trusting rowbytes, as some of the transformations could increase the space needed to hold a row (expand, filler, gray_to_rgb, etc.). See png_read_update_info(), below.

A quick word about text_ptr and num_text. PNG stores comments in keyword/text pairs, one pair per chunk, with no limit on the number of text chunks, and a 2^31 byte limit on their size. While there are suggested keywords, there is no requirement to restrict the use to these strings. It is strongly suggested that keywords and text be sensible to humans (that's the point), so don't use abbreviations. Non-printing symbols are not allowed. See the PNG specification for more details. There is also no requirement to have text after the keyword.

Keywords should be limited to 79 Latin-1 characters without leading or trailing spaces, but non-consecutive spaces are allowed within the keyword. It is possible to have the same keyword any number of times. The text_ptr is an array of png_text structures, each holding a pointer to a language string, a pointer to a keyword and a pointer to a text string. 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 make sure you have read all the text chunks, don't mess with these until after you read the stuff after the image. This will be mentioned again below in the discussion that goes with png_read_end().

Input transformations

After you've read the header information, you can set up the library to handle any special transformations of the image data. The various ways to transform the data will be described in the order that they should occur. This is important, as some of these change the color type and/or bit depth of the data, and some others only work on certain color types and bit depths. Even though each transformation checks to see if it has data that it can do something with, you should make sure to only enable a transformation if it will be valid for the data. For example, don't swap red and blue on grayscale data.

The colors used for the background and transparency values should be supplied in the same format/depth as the current image data. They are stored in the same format/depth as the image data in a bKGD or tRNS chunk, so this is what libpng expects for this data. The colors are transformed to keep in sync with the image data when an application calls the png_read_update_info() routine (see below).

Data will be decoded into the supplied row buffers packed into bytes unless the library has been told to transform it into another format. For example, 4 bit/pixel paletted or grayscale data will be returned 2 pixels/byte with the leftmost pixel in the high-order bits of the byte, unless png_set_packing() is called. 8-bit RGB data will be stored in RGB RGB RGB format unless png_set_filler() 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_strip_16() is called to transform it to regular RGB RGB triplets, or png_set_filler() or png_set_add alpha() is called to insert filler bytes, either before or after each RRGGBB triplet. Similarly, 8-bit or 16-bit grayscale data can be modified with png_set_filler(), png_set_add_alpha(), or png_set_strip_16().

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

RGB, and adds a full alpha channel if there is transparency information in a tRNS chunk. This is most useful on grayscale images with bit depths of 2 or 4 or if there is a multiple-image viewing application that wishes to treat all images in the same way.

```
if (color_type == PNG_COLOR_TYPE_PALETTE)
   png_set_palette_to_rgb(png_ptr);

if (color_type == PNG_COLOR_TYPE_GRAY &&
   bit_depth < 8) png_set_gray_1_2_4_to_8(png_ptr);

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

These three functions are actually aliases for png_set_expand(), added in libpng version 1.0.4, with the function names expanded to improve code readability. In some future version they may actually do different things.

PNG can have files with 16 bits per channel. If you only can handle 8 bits per channel, this will strip the pixels down to 8 bit.

```
if (bit_depth == 16)
   png_set_strip_16(png_ptr);
```

If, for some reason, you don't need the alpha channel on an image, and you want to remove it rather than combining it with the background (but the image author certainly had in mind that you *would* combine it with the background, so that's what you should probably do):

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

In PNG files, the alpha channel in an image is the level of opacity. If you need the alpha channel in an image to be the level of transparency instead of opacity, you can invert the alpha channel (or the tRNS chunk data) after it's read, so that 0 is fully opaque and 255 (in 8-bit or paletted images) or 65535 (in 16-bit images) is fully transparent, with

```
png_set_invert_alpha(png_ptr);
```

PNG files pack pixels of bit depths 1, 2, and 4 into bytes as small as they can, resulting in, for example, 8 pixels per byte for 1 bit files. This code expands to 1 pixel per byte without changing the values of the pixels:

```
if (bit_depth < 8)
   png_set_packing(png_ptr);
```

PNG files have possible bit depths of 1, 2, 4, 8, and 16. All pixels stored in a PNG image have been "scaled" or "shifted" up to the next higher possible bit depth (e.g. from 5 bits/sample in the range [0,31] to 8 bits/sample in the range [0, 255]). However, it is also possible to convert the PNG pixel data back to the original bit depth of the image. This call reduces the pixels back down to the original bit depth:

```
png_color_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 or 16-bit number to fill with, and the location is either PNG_FILLER_BEFORE or PNG_FILLER_AFTER, depending upon whether you want the filler before the RGB or after. This transformation does not affect images that already have full alpha channels. To add an opaque alpha channel, use filler=0xff or 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. This 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_fixed(png_ptr, error_action,
        int red_weight, int green_weight);

  error_action = 1: silently do the conversion
  error_action = 2: issue a warning if the original
                image has any pixel where
                red != green or red != blue
  error_action = 3: issue an error and abort the
                conversion if the original
                image has any pixel where
                red != green or red != blue

  red_weight:       weight of red component times 100000
  green_weight:     weight of green component times 100000
                If either weight is negative, default
                weights (21268, 71514) are used.
```

If you have set error_action = 1 or 2, you can later check whether the image really was gray, after processing the image rows, with the png_get_rgb_to_gray_status(png_ptr) function. It will return a png_byte that is zero if the image was gray or 1 if there were any non-gray pixels. bKGD and sBIT

data will be silently converted to grayscale, using the green channel data, regardless of the error_action setting.

With red_weight+green_weight<=100000, the normalized graylevel is computed:

```
int rw = red_weight * 65536;
int gw = green_weight * 65536;
int bw = 65536 - (rw + gw);
gray = (rw*red + gw*green + bw*blue)/65536;
```

The default values approximate those recommended in the Charles Poynton's Color FAQ, <http://www.inforamp.net/~poynton/> Copyright (c) 1998-01-04 Charles Poynton <poynton at inforamp.net>

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

Libpng approximates this with

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

which can be expressed with integers as

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

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

If you have a grayscale and you are using png_set_expand_depth(), png_set_expand(), or png_set_gray_to_rgb to change to truecolor or to a higher bit-depth, you must either supply the background color as a gray value at the original file bit-depth (need_expand = 1) or else supply the background color as an RGB triplet at the final, expanded bit depth (need_expand = 0). Similarly, if you are reading a paletted image, you must either supply the background color as a palette index (need_expand = 1) or as an RGB triplet that may or may not be in the palette (need_expand = 0).

```
png_color_16 my_background;
png_color_16p image_background;

if (png_get_bKGD(png_ptr, info_ptr, &image_background))
   png_set_background(png_ptr, image_background,
     PNG_BACKGROUND_GAMMA_FILE, 1, 1.0);
else
   png_set_background(png_ptr, &my_background,
     PNG_BACKGROUND_GAMMA_SCREEN, 0, 1.0);
```

The png_set_background() function tells libpng to composite images with alpha or simple transparency against the supplied background color. If the PNG file contains a bKGD chunk (PNG_INFO_bKGD valid), you may use this color, or supply another color more suitable for the current display (e.g., the background color from a web page). You need to tell libpng whether the color is in the gamma space of the display (PNG_BACKGROUND_GAMMA_SCREEN for colors you supply), the file (PNG_BACKGROUND_GAMMA_FILE for colors from the bKGD chunk), or one that is neither of these gammas (PNG_BACKGROUND_GAMMA_UNIQUE - I don't know why anyone would use this, but it's here).

To properly display PNG images on any kind of system, the application needs to know what the display gamma is. Ideally, the user will know this, and the application will allow them to set it. One method of allowing the user to set the display gamma separately for each system is to check for a SCREEN_GAMMA or DISPLAY_GAMMA environment variable, which will hopefully be correctly set.

Note that display_gamma is the overall gamma correction required to produce pleasing results, which

depends on the lighting conditions in the surrounding environment. In a dim or brightly lit room, no compensation other than the physical gamma exponent of the monitor is needed, while in a dark room a slightly smaller exponent is better.

```
  double gamma, screen_gamma;

  if (/* We have a user-defined screen
     gamma value */)
  {
    screen_gamma = user_defined_screen_gamma;
  }
  /* One way that applications can share the same
    screen gamma value */
  else if ((gamma_str = getenv("SCREEN_GAMMA"))
    != NULL)
  {
    screen_gamma = (double)atof(gamma_str);
  }
  /* If we don't have another value */
  else
  {
    screen_gamma = 2.2; /* A good guess for a
        PC monitor in a bright office or a dim room */
    screen_gamma = 2.0; /* A good guess for a
        PC monitor in a dark room */
    screen_gamma = 1.7 or 1.0;  /* A good
        guess for Mac systems */
  }
```

The png_set_gamma() function handles gamma transformations of the data. Pass both the file gamma and the current screen_gamma. If the file does not have a gamma value, you can pass one anyway if you have an idea what it is (usually 0.45455 is a good guess for GIF images on PCs). Note that file gammas are inverted from screen gammas. See the discussions on gamma in the PNG specification for an excellent description of what gamma is, and why all applications should support it. It is strongly recommended that PNG viewers support gamma correction.

```
  if (png_get_gAMA(png_ptr, info_ptr, &gamma))
    png_set_gamma(png_ptr, screen_gamma, gamma);
  else
    png_set_gamma(png_ptr, screen_gamma, 0.45455);
```

If you need to reduce an RGB file to a paletted file, or if a paletted file has more entries then will fit on your screen, png_set_dither() will do that. Note that this is a simple match dither that merely finds the closest color available. This should work fairly well with optimized palettes, and fairly badly with linear color cubes. If you pass a palette that is larger then maximum_colors, the file will reduce the number of colors in the palette so it will fit into maximum_colors. If there is a histogram, it will use it to make more intelligent choices when reducing the palette. If there is no histogram, it may not do as good a job.

```
  if (color_type & PNG_COLOR_MASK_COLOR)
  {
    if (png_get_valid(png_ptr, info_ptr,
      PNG_INFO_PLTE))
    {
      png_uint_16p histogram = NULL;

      png_get_hIST(png_ptr, info_ptr,
        &histogram);
      png_set_dither(png_ptr, palette, num_palette,
```

```
        max_screen_colors, histogram, 1);
    }
    else
    {
      png_color std_color_cube[MAX_SCREEN_COLORS] =
        { ... colors ... };

      png_set_dither(png_ptr, std_color_cube,
        MAX_SCREEN_COLORS, MAX_SCREEN_COLORS,
        NULL,0);
    }
  }
```

PNG files describe monochrome as black being zero and white being one. The following code will reverse this (make black be one and white be zero):

```
  if (bit_depth == 1 && color_type == PNG_COLOR_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_ptr ptr, row_info_ptr
    row_info, png_bytep data)
```

See pngtest.c for a working example. Your function will be called after all of the other transformations have been processed.

You can also set up a pointer to a user structure for use by your callback function, and you can inform libpng that your transform function will change the number of channels or bit depth with the function

```
  png_set_user_transform_info(png_ptr, user_ptr,
    user_depth, user_channels);
```

The user's application, not libpng, is responsible for allocating and freeing any memory required for the

user structure.

You can retrieve the pointer via the function png_get_user_transform_ptr(). For example:

```
voidp read_user_transform_ptr =
  png_get_user_transform_ptr(png_ptr);
```

The last thing to handle is interlacing; this is covered in detail below, but you must call the function here if you want libpng to handle expansion of the interlaced image.

```
number_of_passes = png_set_interlace_handling(png_ptr);
```

After setting the transformations, libpng can update your png_info structure to reflect any transformations you've requested with this call. This is most useful to update the info structure's rowbytes field so you can use it to allocate your image memory. This function will also update your palette with the correct screen_gamma and background if these have been given with the calls above.

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

After you call png_read_update_info(), you can allocate any memory you need to hold the image. The row data is simply raw byte data for all forms of images. As the actual allocation varies among applications, no example will be given. If you are allocating one large chunk, you will need to build an array of pointers to each row, as it will be needed for some of the functions below.

Reading image data

After you've allocated memory, you can read the image data. The simplest way to do this is in one function call. If you are allocating enough memory to hold the whole image, you can just call png_read_image() and libpng will read in all the image data and put it in the memory area supplied. You will need to pass in an array of pointers to each row.

This function automatically handles interlacing, so you don't need to call png_set_interlace_handling() or call this function multiple times, or any of that other stuff necessary with png_read_rows().

```
png_read_image(png_ptr, row_pointers);
```

where row_pointers is:

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

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

If you don't want to read in the whole image at once, you can use png_read_rows() instead. If there is no interlacing (check interlace_type == PNG_INTERLACE_NONE), this is simple:

```
png_read_rows(png_ptr, row_pointers, NULL,
  number_of_rows);
```

where row_pointers is the same as in the png_read_image() call.

If you are doing this just one row at a time, you can do this with a single row_pointer instead of an array of row_pointers:

```
png_bytep row_pointer = row;
png_read_row(png_ptr, row_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) is a somewhat complicated 2D interlace scheme, known as Adam7, that breaks down

an image into seven smaller images of varying size, based on an 8x8 grid.

libpng can fill out those images or it can give them to you "as is". If you want them filled out, there are two ways to do that. The one mentioned in the PNG specification is to expand each pixel to cover those pixels that have not been read yet (the "rectangle" method). This results in a blocky image for the first pass, which gradually smooths out as more pixels are read. The other method is the "sparkle" method, where pixels are drawn only in their final locations, with the rest of the image remaining whatever colors they were initialized to before the start of the read. The first method usually looks better, but tends to be slower, as there are more pixels to put in the rows.

If you don't want libpng to handle the interlacing details, just call png_read_rows() seven times to read in all seven images. Each of the images is a valid image by itself, or they can all be combined on an 8x8 grid to form a single image (although if you intend to combine them you would be far better off using the libpng interlace handling).

The first pass will return an image 1/8 as wide as the entire image (every 8th column starting in column 0) and 1/8 as high as the original (every 8th row starting in row 0), the second will be 1/8 as wide (starting in column 4) and 1/8 as high (also starting in row 0). The third pass will be 1/4 as wide (every 4th pixel starting in column 0) and 1/8 as high (every 8th row starting in row 4), and the fourth pass will be 1/4 as wide and 1/4 as high (every 4th column starting in column 2, and every 4th row starting in row 0). The fifth pass will return an image 1/2 as wide, and 1/4 as high (starting at column 0 and row 2), while the sixth pass will be 1/2 as wide and 1/2 as high as the original (starting in column 1 and row 0). The seventh and final pass will be as wide as the original, and 1/2 as high, containing all of the odd numbered scanlines. Phew!

If you want libpng to expand the images, call this before calling png_start_read_image() or png_read_update_info():

```
if (interlace_type == PNG_INTERLACE_ADAM7)
   number_of_passes
     = png_set_interlace_handling(png_ptr);
```

This will return the number of passes needed. Currently, this is seven, but may change if another interlace type is added. This function can be called even if the file is not interlaced, where it will return one pass.

If you are not going to display the image after each pass, but are going to wait until the entire image is read in, use the sparkle effect. This effect is faster and the end result of either method is exactly the same. If you are planning on displaying the image after each pass, the "rectangle" effect is generally considered the better looking one.

If you only want the "sparkle" effect, just call png_read_rows() as normal, with the third parameter NULL. Make sure you make pass over the image number_of_passes times, and you don't change the data in the rows between calls. You can change the locations of the data, just not the data. Each pass only writes the pixels appropriate for that pass, and assumes the data from previous passes is still valid.

```
png_read_rows(png_ptr, row_pointers, NULL,
  number_of_rows);
```

If you only want the first effect (the rectangles), do the same as before except pass the row buffer in the third parameter, and leave the second parameter NULL.

```
png_read_rows(png_ptr, NULL, row_pointers,
  number_of_rows);
```

Finishing a sequential read

After you are finished reading the image through either the high- or low-level interfaces, you can finish reading the file. If you are interested in comments or time, which may be stored either before or after

the image data, you should pass the separate png_info struct if you want to keep the comments from before and after the image separate. If you are not interested, you can pass NULL.

```
png_read_end(png_ptr, end_info);
```

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

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

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

```
png_free_data(png_ptr, info_ptr, mask, seq)
mask - identifies data to be freed, a mask
      containing the logical 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_zalloc() and passed in via a png_set_*() function, with

```
png_data_freer(png_ptr, info_ptr, freer, mask)
mask   - which data elements are affected
        same choices as in png_free_data()
freer  - one of
          PNG_DESTROY_WILL_FREE_DATA
          PNG_SET_WILL_FREE_DATA
          PNG_USER_WILL_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_zalloc() 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 logical 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_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.

Reading PNG files progressively

The progressive reader is slightly different then the non-progressive reader. Instead of calling png_read_info(), png_read_rows(), and png_read_end(), you make one call to png_process_data(), which calls callbacks when it has the info, a row, or the end of the image. You set up these callbacks with png_set_progressive_read_fn(). You don't have to worry about the input/output functions of libpng, as you are giving the library the data directly in png_process_data(). I will assume that you have read the section on reading PNG files above, so I will only highlight the differences (although I will show all of the code).

```
png_structp png_ptr; png_infop info_ptr;

 /*  An example code fragment of how you would
     initialize the progressive reader in your
     application. */
 int
 initialize_png_reader()
 {
   png_ptr = png_create_read_struct
      (PNG_LIBPNG_VER_STRING, (png_voidp)user_error_ptr,
       user_error_fn, user_warning_fn);
   if (!png_ptr)
      return (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 then 256 bytes
      yet).  When this function returns, you may
      want to display any rows that were generated
      in the row callback if you don't already do
      so there.
    */
   png_process_data(png_ptr, info_ptr, buffer, length);
   return 0;
}

/* This function is called (as set by
   png_set_progressive_read_fn() above) when enough data
   has been supplied so all of the header has been
   read.
*/
void
info_callback(png_structp png_ptr, png_infop info)
{
```

```
   /* Do any setup here, including setting any of
      the transformations mentioned in the Reading
      PNG files section.  For now, you _must_ call
      either png_start_read_image() or
      png_read_update_info() after all the
      transformations are set (even if you don't set
      any).  You may start getting rows before
      png_process_data() returns, so this is your
      last chance to prepare for that.
    */
}

/* This function is called when each row of image
   data is complete */
void
row_callback(png_structp png_ptr, png_bytep new_row,
   png_uint_32 row_num, int pass)
{
   /* If the image is interlaced, and you turned
      on the interlace handler, this function will
      be called for every row in every pass.  Some
      of these rows will not be changed from the
      previous pass.  When the row is not changed,
      the new_row variable will be NULL.  The rows
      and passes are called in order, so you don't
      really need the row_num and pass, but I'm
      supplying them because it may make your life
      easier.

      For the non-NULL rows of interlaced images,
      you must call png_progressive_combine_row()
      passing in the row and the old row.  You can
      call this function for NULL rows (it will just
      return) and for non-interlaced images (it just
      does the memcpy for you) if it will make the
      code easier.  Thus, you can just do this for
      all cases:
    */

       png_progressive_combine_row(png_ptr, old_row,
         new_row);

   /* where old_row is what was displayed for
      previously for the row.  Note that the first
      pass (pass == 0, really) will completely cover
      the old row, so the rows do not have to be
      initialized.  After the first pass (and only
      for interlaced images), you will have to pass
      the current row, and the function will combine
      the old row and the new row.
   */
}

void
end_callback(png_structp png_ptr, png_infop info)
{
   /* This function is called after the whole image
      has been read, including any chunks after the
      image (up to and including the IEND).  You
```

```
    will usually have the same info chunk as you
    had in the header, although some data may have
    been added to the comments and time fields.

    Most people won't do much here, perhaps setting
    a flag that marks the image as finished.
  */
}
```

IV. Writing

Much of this is very similar to reading. However, everything of importance is repeated here, so you won't have to constantly look back up in the reading section to understand writing.

Setup

You will want to do the I/O initialization before you get into libpng, so if it doesn't work, you don't have anything to undo. If you are not using the standard I/O functions, you will need to replace them with custom writing functions. See the discussion under Customizing libpng.

```
FILE *fp = fopen(file_name, "wb");
if (!fp)
{
  return (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_SETJMP_NOT_SUPPORTED, in which case errors will result in a call to PNG_ABORT() which defaults to abort().

Now you need to set up the output code. The default for libpng is to use the C function fwrite(). If you use this, you will need to pass a valid FILE * in the function png_init_io(). Be sure that the file is opened in binary mode. Again, if you wish to handle writing data in another way, see the discussion on libpng I/O handling in the Customizing Libpng section below.

```
png_init_io(png_ptr, fp);
```

Write callbacks

At this point, you can set up a callback function that will be called after each row has been written, which you can use to control a progress meter or the like. It's demonstrated in pngtest.c. You must supply a function

```
void write_row_callback(png_ptr, png_uint_32 row,
  int pass);
{
 /* put your code here */
}
```

(You can give it another name that you like instead of "write_row_callback")

To inform libpng about your function, use

```
png_set_write_status_fn(png_ptr, write_row_callback);
```

You now have the option of modifying how the compression library will run. The following functions are mainly for testing, but may be useful in some cases, like if you need to write PNG files extremely fast and are willing to give up some compression, or if you want to get the maximum possible compression at the expense of slower writing. If you have no special needs in this area, let the library do what it wants by not calling this function at all, as it has been tuned to deliver a good speed/compression ratio. The second parameter to png_set_filter() is the filter method, for which the only valid 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 logical 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_AVE   | PNG_FILTER_VALUE_AVE  |
   PNG_FILTER_PAETH | PNG_FILTER_VALUE_PAETH|
   PNG_ALL_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.

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

/* set other zlib parameters */
png_set_compression_mem_level(png_ptr, 8);
png_set_compression_strategy(png_ptr,
   Z_DEFAULT_STRATEGY);
png_set_compression_window_bits(png_ptr, 15);
png_set_compression_method(png_ptr, 8);
png_set_compression_buffer_size(png_ptr, 8192)

extern PNG_EXPORT(void,png_set_zbuf_size)
```

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)

png_set_PLTE(png_ptr, info_ptr, palette,
num_palette);
palette - the palette for the file
(array of png_color)
num_palette - number of entries in the palette

png_set_gAMA(png_ptr, info_ptr, gamma);
gamma - the gamma the image was created
at (PNG_INFO_gAMA)

png_set_sRGB(png_ptr, info_ptr, srgb_intent);
srgb_intent - the rendering intent
(PNG_INFO_sRGB) The presence of
the sRGB chunk means that the pixel
data is in the sRGB color space.
This chunk also implies specific
values of gAMA and cHRM. Rendering
intent is the CSS-1 property that
has been defined by the International
Color Consortium
(http://www.color.org).
It can be one of
PNG_sRGB_INTENT_SATURATION,
PNG_sRGB_INTENT_PERCEPTUAL,
PNG_sRGB_INTENT_ABSOLUTE, or
PNG_sRGB_INTENT_RELATIVE.

png_set_sRGB_gAMA_and_cHRM(png_ptr, info_ptr,
srgb_intent);
srgb_intent - the rendering intent
(PNG_INFO_sRGB) The presence of the
sRGB chunk means that the pixel

data is in the sRGB color space.
This function also causes gAMA and
cHRM chunks with the specific values
that are consistent with sRGB to be
written.

```
png_set_iCCP(png_ptr, info_ptr, name, compression_type,
             profile, proflen);
name           - The profile name.
compression    - The compression type; always
             PNG_COMPRESSION_TYPE_BASE for PNG 1.0.
             You may give NULL to this argument to
             ignore it.
profile        - International Color Consortium color
             profile data. May contain NULs.
proflen        - length of profile data in bytes.

png_set_sBIT(png_ptr, info_ptr, sig_bit);
sig_bit        - the number of significant bits for
             (PNG_INFO_sBIT) each of the gray, red,
             green, and blue channels, whichever are
             appropriate for the given color type
             (png_color_16)

png_set_tRNS(png_ptr, info_ptr, trans, num_trans,
  trans_values);
trans          - array of transparent entries for
             palette (PNG_INFO_tRNS)
trans_values   - graylevel or color sample values of
             the single transparent color for
             non-paletted images (PNG_INFO_tRNS)
num_trans      - number of transparent entries
             (PNG_INFO_tRNS)

png_set_hIST(png_ptr, info_ptr, hist);
            (PNG_INFO_hIST)
hist           - histogram of palette (array of
             png_uint_16)

png_set_tIME(png_ptr, info_ptr, mod_time);
mod_time       - time image was last modified
             (PNG_VALID_tIME)

png_set_bKGD(png_ptr, info_ptr, background);
background     - background color (PNG_VALID_bKGD)

png_set_text(png_ptr, info_ptr, text_ptr, num_text);
text_ptr       - array of png_text holding image
             comments
text_ptr[i].compression - type of compression used
          on "text" PNG_TEXT_COMPRESSION_NONE
                PNG_TEXT_COMPRESSION_zTXt
                PNG_ITXT_COMPRESSION_NONE
                PNG_ITXT_COMPRESSION_zTXt
text_ptr[i].key   - keyword for comment.  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).
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
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 1000 bytes, it is not worth compressing it. After the text has been written out to the file, the compression type is set to PNG_TEXT_COMPRESSION_NONE_WR or PNG_TEXT_COMPRESSION_zTXt_WR, so that it isn't written out again at the end (in case you are calling png_write_end() with the same struct.

The keywords that are given in the PNG Specification are:

Title	Short (one line) title or caption for image
Author	Name of image's creator
Description	Description of image (possibly long)
Copyright	Copyright notice
Creation Time	Time of original image creation (usually RFC 1123 format, see below)
Software	Software used to create the image
Disclaimer	Legal disclaimer
Warning	Warning of nature of content
Source	Device used to create the image
Comment	Miscellaneous comment; conversion from other image format

The keyword-text pairs work like this. Keywords should be short simple descriptions of what the comment is about. Some typical keywords are found in the PNG specification, as is some recommendations on keywords. You can repeat keywords in a file. You can even write some text before the image and some after. For example, you may want to put a description of the image before the image, but leave the disclaimer until after, so viewers working over modem connections don't have to wait for the disclaimer to go over the modem before they start seeing the image. Finally, keywords should be full words, not abbreviations. Keywords and text are in the ISO 8859-1 (Latin-1) character set (a superset of regular ASCII) and can not contain NUL characters, and should not contain control or other unprintable characters. To make the comments widely readable, stick with basic ASCII, and avoid machine specific character set extensions like the IBM-PC character set. The keyword must be present, but you can leave off the text string on non-compressed pairs. Compressed pairs must have a text string, as only the text string is compressed anyway, so the compression would be meaningless.

PNG supports modification time via the png_time structure. Two conversion routines are 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(png_timep) is provided to convert from PNG time to an RFC 1123 format string.

Writing unknown chunks

You can use the png_set_unknown_chunks function to queue up chunks for writing. You give it a chunk name, raw data, and a size; that's all there is to it. The chunks will be written by the next following png_write_info_before_PLTE, png_write_info, or png_write_end function. Any chunks previously read into the info structure's unknown-chunk list will also be written out in a sequence that satisfies the PNG specification's ordering rules.

The high-level write interface

At this point there are two ways to proceed; through the high-level write interface, or through a sequence of low-level write operations. You can use the high-level interface if your image data is present 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.
```

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 logical 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().

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_ptr ptr, row_info_ptr
  row_info, png_bytep data)
```

See pngtest.c for a working example. Your function will be called before any of the other transformations are processed.

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

```
png_set_user_transform_info(png_ptr, user_ptr, 0, 0);
```

The user_channels and user_depth parameters of this function are ignored when writing; you can set them to zero as shown.

You can retrieve the pointer via the function png_get_user_transform_ptr(). For example:

```
voidp write_user_transform_ptr =
  png_get_user_transform_ptr(png_ptr);
```

It is possible to have libpng flush any pending output, either manually, or automatically after a certain number of lines have been written. To flush the output stream a single time call:

```
png_write_flush(png_ptr);
```

and to have libpng flush the output stream periodically after a certain number of scanlines have been written, call:

```
png_set_flush(png_ptr, nrows);
```

Note that the distance between rows is from the last time png_write_flush() was called, or the first row of the image if it has never been called. So if you write 50 lines, and then png_set_flush 25, it will flush the output on the next scanline, and every 25 lines thereafter, unless png_write_flush() is called before 25 more lines have been written. If nrows is too small (less than about 10 lines for a 640 pixel wide RGB image) the image compression may decrease noticeably (although this may be acceptable for real-time applications). Infrequent flushing will only degrade the compression performance by a few percent over images that do not use flushing.

Writing the image data

That's it for the transformations. Now you can write the image data. The simplest way to do this is in one function call. If you have the whole image in memory, you can just call png_write_image() and libpng will write the image. You will need to pass in an array of pointers to each row. This function automatically handles interlacing, so you don't need to call png_set_interlace_handling() or call this function multiple times, or any of that other stuff necessary with png_write_rows().

```
png_write_image(png_ptr, row_pointers);
```

where row_pointers is:

```
png_byte *row_pointers[height];
```

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

If you don't want to write the whole image at once, you can use png_write_rows() instead. If the file is not interlaced, this is simple:

```
png_write_rows(png_ptr, row_pointers,
  number_of_rows);
```

row_pointers is the same as in the png_write_image() call.

If you are just writing one row at a time, you can do this with a single row_pointer instead of an array of row_pointers:

```
png_bytep row_pointer = row;

png_write_row(png_ptr, row_pointer);
```

When the file is interlaced, things can get a good deal more complicated. The only currently (as of 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 seven sub-images.

If you want libpng to build the sub-images, call this before you start writing any rows:

```
number_of_passes =
  png_set_interlace_handling(png_ptr);
```

This will return the number of passes needed. Currently, this is seven, but may change if another interlace type is added.

Then write the complete image number_of_passes times.

```
png_write_rows(png_ptr, row_pointers,
  number_of_rows);
```

As some of these rows are not used, and thus return immediately, you may want to read about interlacing in the PNG specification, and only update the rows that are actually used.

Finishing a sequential write

After you are finished writing the image, you should finish writing the file. If you are interested in writing comments or time, you should pass an appropriately filled png_info pointer. If you are not interested, you can pass NULL.

```
png_write_end(png_ptr, info_ptr);
```

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

```
png_destroy_write_struct(&png_ptr, &info_ptr);
```

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

```
png_free_data(png_ptr, info_ptr, mask, seq)
mask  - identifies data to be freed, a mask
      containing the logical 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_zalloc() and passed in via a png_set_*() function, with

```
png_data_freer(png_ptr, info_ptr, freer, mask)
mask   - which data elements are affected
       same choices as in png_free_data()
freer  - one of
         PNG_DESTROY_WILL_FREE_DATA
         PNG_SET_WILL_FREE_DATA
         PNG_USER_WILL_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_zalloc() 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. Modifying/Customizing libpng:

There are three 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. The third is a run-time issue: choosing between and/or tuning one or more alternate versions of computationally intensive routines; specifically, optimized assembly-language (and therefore compiler- and platform-dependent) versions.

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() and png_free(). These currently just call the standard C functions. If your pointers can't access more then 64K at a time, you will want to set MAXSEG_64K in zlib.h. Since it is unlikely that the method of handling memory allocation on a platform will change between applications, these functions must be modified in the library at compile time. 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_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().

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, png_size_t length);
void user_write_data(png_structp png_ptr,
   png_bytep data, png_size_t length);
void user_flush_data(png_structp png_ptr);
```

Supplying NULL for the read, write, or flush functions sets them back to using the default C stream functions. It is an error to read from a write stream, and vice versa.

Error handling in libpng is done through png_error() and png_warning(). Errors handled through png_error() are fatal, meaning that png_error() should never return to its caller. Currently, this is handled via setjmp() and longjmp() (unless you have compiled libpng with PNG_SETJMP_NOT_SUPPORTED, in which case it is handled via PNG_ABORT()), but you could change this to do things like exit() if you should wish.

On non-fatal errors, png_warning() is called to print a warning message, and then control returns to the calling code. By default png_error() and png_warning() print a message on stderr via fprintf() unless the library is compiled with PNG_NO_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);

png_voidp error_ptr = png_get_error_ptr(png_ptr);
```

If NULL is supplied for either error_fn or warning_fn, then the libpng default function will be used, calling fprintf() and/or longjmp() if a problem is encountered. The replacement error functions should have parameters as follows:

```
void user_error_fn(png_structp png_ptr,
  png_const_charp error_msg);
void user_warning_fn(png_structp png_ptr,
  png_const_charp warning_msg);
```

The motivation behind using setjmp() and longjmp() is the C++ throw and catch exception handling methods. This makes the code much easier to write, as there is no need to check every return code of every function call. However, there are some uncertainties about the status of local variables after a longjmp, so the user may want to be careful about doing anything after setjmp returns non-zero besides returning itself. Consult your compiler documentation for more details. For an alternative approach, you may wish to use the "cexcept" facility (see http://cexcept.sourceforge.net).

Custom chunks

If you need to read or write custom chunks, you may need to get deeper into the libpng code. The library now has mechanisms for storing and writing chunks of unknown type; you can even declare callbacks for custom chunks. Hoewver, this may not be good enough if the library code itself needs to know about interactions between your chunk and existing ‘intrinsic’ chunks.

If you need to write a new intrinsic chunk, first read the PNG specification. Acquire a first level of understanding of how it works. Pay particular attention to the sections that describe chunk names, and look at how other chunks were designed, so you can do things similarly. Second, check out the sections of libpng that read and write chunks. Try to find a chunk that is similar to yours and use it as a template. More details can be found in the comments inside the code. It is best to handle unknown chunks in a generic method, via callback functions, instead of by modifying libpng functions.

If you wish to write your own transformation for the data, look through the part of the code that does the transformations, and check out some of the simpler ones to get an idea of how they work. Try to find a similar transformation to the one you want to add and copy off of it. More details can be found in the comments inside the code itself.

Configuring for 16 bit platforms

You will want to look into zconf.h to tell zlib (and thus libpng) that it cannot allocate more then 64K at a time. Even if you can, the memory won’t be accessible. So limit zlib and libpng to 64K by defining MAXSEG_64K.

Configuring for DOS

For DOS users who only have access to the lower 640K, you will have to limit zlib’s memory usage via a png_set_compression_mem_level() call. See zlib.h or zconf.h in the zlib library for more information.

Configuring for Medium Model

Libpng’s support for medium model has been tested on most of the popular compilers. Make sure MAXSEG_64K gets defined, USE_FAR_KEYWORD gets defined, and FAR gets defined to far in pngconf.h, and you should be all set. Everything in the library (except for zlib’s structure) is expecting far data. You must use the typedefs with the p or pp on the end for pointers (or at least look at them and be careful). Make note that the rows of data are defined as png_bytepp, which is an unsigned char far * far *.

Configuring for gui/windowing platforms:

You will need to write new error and warning functions that use the GUI interface, as described previously, and set them to be the error and warning functions at the time that png_create_*_struct() is called, in order to have them available during the structure initialization. They can be changed later via png_set_error_fn(). On some compilers, you may also have to change the memory allocators

(png_malloc, etc.).

Configuring for compiler xxx:

All includes for libpng are in pngconf.h. If you need to add/change/delete an include, this is the place to do it. The includes that are not needed outside libpng are protected by the PNG_INTERNAL definition, which is only defined for those routines inside libpng itself. The files in libpng proper only include png.h, which includes pngconf.h.

Configuring zlib:

There are special functions to configure the compression. Perhaps the most useful one changes the compression level, which currently uses input compression values in the range 0 - 9. The library normally uses the default compression level (Z_DEFAULT_COMPRESSION = 6). Tests have shown that for a large majority of images, compression values in the range 3-6 compress nearly as well as higher levels, and do so much faster. For online applications it may be desirable to have maximum speed (Z_BEST_SPEED = 1). With versions of zlib after v0.99, you can also specify no compression (Z_NO_COMPRESSION = 0), but this would create files larger than just storing the raw bitmap. You can specify the compression level by calling:

```
png_set_compression_level(png_ptr, level);
```

Another useful one is to reduce the memory level used by the library. The memory level defaults to 8, but it can be lowered if you are short on memory (running DOS, for example, where you only have 640K). 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.

```
png_set_compression_mem_level(png_ptr, level);
```

The other functions are for configuring zlib. They are not recommended for normal use and may result in writing an invalid PNG file. See zlib.h for more information on what these mean.

```
png_set_compression_strategy(png_ptr,
   strategy);
png_set_compression_window_bits(png_ptr,
   window_bits);
png_set_compression_method(png_ptr, method);
png_set_compression_buffer_size(png_ptr, size);
```

Controlling row filtering

If you want to control whether libpng uses filtering or not, which filters are used, and how it goes about picking row filters, you can call one of these functions. The selection and configuration of row filters can have a significant impact on the size and encoding speed and a somewhat lesser impact on the decoding speed of an image. Filtering is enabled by default for RGB and grayscale images (with and without alpha), but not for paletted images nor for any images with bit depths less than 8 bits/pixel.

The 'method' parameter sets the main filtering method, which is currently only '0' in the PNG 1.2 specification. The 'filters' parameter sets which filter(s), if any, should be used for each scanline. Possible values are PNG_ALL_FILTERS and PNG_NO_FILTERS to turn filtering on and off, respectively.

Individual filter types are PNG_FILTER_NONE, PNG_FILTER_SUB, PNG_FILTER_UP, PNG_FILTER_AVG, PNG_FILTER_PAETH, which can be bitwise ORed together 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_FILTER_NONE | PNG_FILTER_SUB
      PNG_FILTER_UP | PNG_FILTER_AVE |
      PNG_FILTER_PAETH | PNG_ALL_FILTERS;

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().
```

It is also possible to influence how libpng chooses from among the available filters. This is done in one or both of two ways - by telling it how important it is to keep the same filter for successive rows, and by telling it the relative computational costs of the filters.

```
double weights[3] = {1.5, 1.3, 1.1},
  costs[PNG_FILTER_VALUE_LAST] =
  {1.0, 1.3, 1.3, 1.5, 1.7};

png_set_filter_heuristics(png_ptr,
  PNG_FILTER_HEURISTIC_WEIGHTED, 3,
  weights, costs);
```

The weights are multiplying factors that indicate to libpng that the row filter should be the same for successive rows unless another row filter is that many times better than the previous filter. In the above example, if the previous 3 filters were SUB, SUB, NONE, the SUB filter could have a "sum of absolute differences" 1.5 x 1.3 times higher than other filters and still be chosen, while the NONE filter could have a sum 1.1 times higher than other filters and still be chosen. Unspecified weights are taken to be 1.0, and the specified weights should probably be declining like those above in order to emphasize recent filters over older filters.

The filter costs specify for each filter type a relative decoding cost to be considered when selecting row filters. This means that filters with higher costs are less likely to be chosen over filters with lower costs, unless their "sum of absolute differences" is that much smaller. The costs do not necessarily reflect the exact computational speeds of the various filters, since this would unduly influence the final image size.

Note that the numbers above were invented purely for this example and are given only to help explain the function usage. Little testing has been done to find optimum values for either the costs or the weights.

Removing unwanted object code

There are a bunch of #define's in pngconf.h that control what parts of libpng are compiled. All the defines end in _SUPPORTED. If you are never going to use a capability, you can change the #define to #undef before recompiling libpng and save yourself code and data space, or you can turn off individual capabilities with defines that begin with PNG_NO_.

You can also turn all of the transforms and ancillary chunk capabilities off en masse with compiler directives that define PNG_NO_READ[or WRITE]_TRANSFORMS, or PNG_NO_READ[or WRITE]_ANCILLARY_CHUNKS, or all four, along with directives to turn on any of the capabilities that you do want. The PNG_NO_READ[or WRITE]_TRANSFORMS directives disable the extra transformations but still leave the library fully capable of reading and writing PNG files with all known public chunks Use of the PNG_NO_READ[or WRITE]_ANCILLARY_CHUNKS directive produces a library that is incapable of reading or writing ancillary chunks. If you are not using the progressive reading capability, you can turn that off with PNG_NO_PROGRESSIVE_READ (don't confuse this with the INTERLACING capability, which you'll still have).

All the reading and writing specific code are in separate files, so the linker should only grab the files it needs. However, if you want to make sure, or if you are building a stand alone library, all the reading files start with pngr and all the writing files start with pngw. The files that don't match either (like png.c, pngtrans.c, etc.) are used for both reading and writing, and always need to be included. The progressive reader is in pngpread.c

If you are creating or distributing a dynamically linked library (a .so or DLL file), you should not remove or disable any parts of the library, as this will cause applications linked with different versions of the library to fail if they call functions not available in your library. The size of the library itself should not be an issue, because only those sections that are actually used will be loaded into memory.

Requesting debug printout

The macro definition PNG_DEBUG can be used to request debugging printout. Set it to an integer value in the range 0 to 3. Higher numbers result in increasing amounts of debugging information. The information is printed to the "stderr" file, unless another file name is specified in the PNG_DEBUG_FILE macro definition.

When PNG_DEBUG > 0, the following functions (macros) become available:

```
png_debug(level, message)
png_debug1(level, message, p1)
png_debug2(level, message, p1, p2)
```

in which "level" is compared to PNG_DEBUG to decide whether to print the message, "message" is the formatted string to be printed, and p1 and p2 are parameters that are to be embedded in the string according to printf-style formatting directives. For example,

```
png_debug1(2, "foo=%d0, foo);
```

is expanded to

```
if(PNG_DEBUG > 2)
  fprintf(PNG_DEBUG_FILE, "foo=%d0, foo);
```

When PNG_DEBUG is defined but is zero, the macros aren't defined, but you can still use PNG_DEBUG to control your own debugging:

```
#ifdef PNG_DEBUG
   fprintf(stderr, ...
#endif
```

When PNG_DEBUG = 1, the macros are defined, but only png_debug statements having level = 0 will be printed. There aren't any such statements in this version of libpng, but if you insert some they will be printed.

VI. Runtime optimization

A new feature in libpng 1.2.0 is the ability to dynamically switch between standard and optimized versions of some routines. Currently these are limited to three computationally intensive tasks when reading PNG files: decoding row filters, expanding interlacing, and combining interlaced or transparent row data with previous row data. Currently the optimized versions are available only for x86 (Intel, AMD, etc.) platforms with MMX support, though this may change in future versions. (For example, the non-MMX assembler optimizations for zlib might become similarly runtime-selectable in future releases, in which case libpng could be extended to support them. Alternatively, the compile-time choice of floating-point versus integer routines for gamma correction might become runtime-selectable.)

Because such optimizations tend to be very platform- and compiler-dependent, both in how they are written and in how they perform, the new runtime code in libpng has been written to allow programs to query, enable, and disable either specific optimizations or all such optimizations. For example, to

enable all possible optimizations (bearing in mind that some "optimizations" may actually run more slowly in rare cases):

```
#if defined(PNG_LIBPNG_VER) && (PNG_LIBPNG_VER >= 10200)
  png_uint_32 mask, flags;

  flags = png_get_asm_flags(png_ptr);
  mask = png_get_asm_flagmask(PNG_SELECT_READ | PNG_SELECT_WRITE);
  png_set_asm_flags(png_ptr, flags | mask);
#endif
```

To enable only optimizations relevant to reading PNGs, use PNG_SELECT_READ by itself when calling png_get_asm_flagmask(); similarly for optimizing only writing. To disable all optimizations:

```
#if defined(PNG_LIBPNG_VER) && (PNG_LIBPNG_VER >= 10200)
  flags = png_get_asm_flags(png_ptr);
  mask = png_get_asm_flagmask(PNG_SELECT_READ | PNG_SELECT_WRITE);
  png_set_asm_flags(png_ptr, flags & ~mask);
#endif
```

To enable or disable only MMX-related features, use png_get_mmx_flagmask() in place of png_get_asm_flagmask(). The mmx version takes one additional parameter:

```
#if defined(PNG_LIBPNG_VER) && (PNG_LIBPNG_VER >= 10200)
  int selection = PNG_SELECT_READ | PNG_SELECT_WRITE;
  int compilerID;

  mask = png_get_mmx_flagmask(selection, &compilerID);
#endif
```

On return, compilerID will indicate which version of the MMX assembler optimizations was compiled. Currently two flavors exist: Microsoft Visual C++ (compilerID == 1) and GNU C (a.k.a. gcc/gas, compilerID == 2). On non-x86 platforms or on systems compiled without MMX optimizations, a value of -1 is used.

Note that both png_get_asm_flagmask() and png_get_mmx_flagmask() return all valid, settable optimization bits for the version of the library that's currently in use. In the case of shared (dynamically linked) libraries, this may include optimizations that did not exist at the time the code was written and compiled. It is also possible, of course, to enable only known, specific optimizations; for example:

```
#if defined(PNG_LIBPNG_VER) && (PNG_LIBPNG_VER >= 10200)
  flags = 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_set_asm_flags(png_ptr, flags);
#endif
```

This method would enable only the MMX read-optimizations available at the time of libpng 1.2.0's release, regardless of whether a later version of the DLL were actually being used. (Also note that these functions did not exist in versions older than 1.2.0, so any attempt to run a dynamically linked app on such an older version would fail.)

To determine whether the processor supports MMX instructions at all, use the png_mmx_support() function:

```
#if defined(PNG_LIBPNG_VER) && (PNG_LIBPNG_VER >= 10200)
  mmxsupport = png_mmx_support();
```

```
#endif
```

It returns -1 if MMX support is not compiled into libpng, 0 if MMX code is compiled but MMX is not supported by the processor, or 1 if MMX support is fully available. Note that png_mmx_support(), png_get_mmx_flagmask(), and png_get_asm_flagmask() all may be called without allocating and initializing any PNG structures (for example, as part of a usage screen or "about" box).

The following code can be used to prevent an application from using the thread_unsafe features, even if libpng was built with PNG_THREAD_UNSAFE_OK defined:

```
#if defined(PNG_USE_PNGGCCRD) && defined(PNG_ASSEMBLER_CODE_SUPPORTED) &&
defined(PNG_THREAD_UNSAFE_OK)
  /* Disable thread-unsafe features of pnggccrd */
  if (png_access_version() >= 10200)
  {
    png_uint_32 mmx_disable_mask = 0;
    png_uint_32 asm_flags;

    mmx_disable_mask |= ( PNG_ASM_FLAG_MMX_READ_COMBINE_ROW |
PNG_ASM_FLAG_MMX_READ_FILTER_SUB |
PNG_ASM_FLAG_MMX_READ_FILTER_AVG |
PNG_ASM_FLAG_MMX_READ_FILTER_PAETH );
    asm_flags = png_get_asm_flags(png_ptr);
    png_set_asm_flags(png_ptr, asm_flags & ~mmx_disable_mask);
  } #endif
```

For more extensive examples of runtime querying, enabling and disabling of optimized features, see contrib/gregbook/readpng2.c in the libpng source-code distribution.

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 logical 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 logical 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 http://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 2.0.0.

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.

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).

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

```
png_uint_32 application_vn = PNG_LIBPNG_VER;
```

IX. Y2K Compliance in libpng

December 3, 2004

Since the PNG Development group is an ad-hoc body, we can't make an official declaration.

This is your unofficial assurance that libpng from version 0.71 and upward through 1.2.8 are Y2K compliant. It is my belief that earlier versions were also Y2K compliant.

Libpng only has three year fields. One is a 2-byte unsigned integer that will hold years up to 65535. The other two hold the date in text format, and will hold years up to 9999.

The integer is
"png_uint_16 year" in png_time_struct.

The strings are
"png_charp time_buffer" in png_struct and
"near_time_buffer", which is a local character string in png.c.

There are seven time-related functions:

```
png_convert_to_rfc_1123() in png.c
  (formerly png_convert_to_rfc_1152() in error)
png_convert_from_struct_tm() in pngwrite.c, called
  in pngwrite.c
png_convert_from_time_t() in pngwrite.c
png_get_tIME() in pngget.c
png_handle_tIME() in pngrutil.c, called in pngread.c
png_set_tIME() in pngset.c
png_write_tIME() in pngwutil.c, called in pngwrite.c
```

All appear to handle dates properly in a Y2K environment. The png_convert_from_time_t() function calls gmtime() to convert from system clock time, which returns (year - 1900), which we properly convert to the full 4-digit year. There is a possibility that applications using libpng are not passing 4-digit years into the png_convert_to_rfc_1123() function, or that they are incorrectly passing only a 2-digit year instead of "year - 1900" into the png_convert_from_struct_tm() function, but this is not under our control. The libpng documentation has always stated that it works with 4-digit years, and the APIs have been documented as such.

The tIME chunk itself is also Y2K compliant. It uses a 2-byte unsigned integer to hold the year, and can hold years as large as 65535.

zlib, upon which libpng depends, is also Y2K compliant. It contains no date-related code.

Glenn Randers-Pehrson
libpng maintainer
PNG Development Group

NOTE

Note about libpng version numbers:

Due to various miscommunications, unforeseen code incompatibilities and occasional factors outside the authors' control, version numbering on the library has not always been consistent and straightforward. The following table summarizes matters since version 0.89c, which was the first widely used release:

source version	png.h string	png.h int	shared-lib version
0.89c ("beta 3")	0.89	89	1.0.89
0.90 ("beta 4")	0.90	90	0.90
0.95 ("beta 5")	0.95	95	0.95
0.96 ("beta 6")	0.96	96	0.96
0.97b ("beta 7")	1.00.97	97	1.0.1
0.97c	0.97	97	2.0.97
0.98	0.98	98	2.0.98
0.99	0.99	98	2.0.99
0.99a-m	0.99	99	2.0.99
1.00	1.00	100	2.1.0
1.0.0	1.0.0	100	2.1.0
1.0.0 (from here on, the		100	2.1.0
1.0.1 png.h string is		10001	2.1.0
1.0.1a-e identical to the		10002	from here on, the
1.0.2 source version)		10002	shared library is 2.V
1.0.2a-b		10003	where V is the source
1.0.1		10001	code version except as
1.0.1a-e		10002	2.1.0.1a-e noted.
1.0.2		10002	2.1.0.2
1.0.2a-b		10003	2.1.0.2a-b
1.0.3		10003	2.1.0.3
1.0.3a-d		10004	2.1.0.3a-d
1.0.4		10004	2.1.0.4
1.0.4a-f		10005	2.1.0.4a-f
1.0.5 (+ 2 patches)		10005	2.1.0.5
1.0.5a-d		10006	2.1.0.5a-d
1.0.5e-r		10100	2.1.0.5e-r
1.0.5s-v		10006	2.1.0.5s-v
1.0.6 (+ 3 patches)		10006	2.1.0.6

1.0.6d-g		10007	2.1.0.6d-g
1.0.6h		10007	10.6h
1.0.6i		10007	10.6i
1.0.6j		10007	2.1.0.6j
1.0.7beta11-14	DLLNUM	10007	2.1.0.7beta11-14
1.0.7beta15-18	1	10007	2.1.0.7beta15-18
1.0.7rc1-2	1	10007	2.1.0.7rc1-2
1.0.7	1	10007	2.1.0.7
1.0.8beta1-4	1	10008	2.1.0.8beta1-4
1.0.8rc1	1	10008	2.1.0.8rc1
1.0.8	1	10008	2.1.0.8
1.0.9beta1-6	1	10009	2.1.0.9beta1-6
1.0.9rc1	1	10009	2.1.0.9rc1
1.0.9beta7-10	1	10009	2.1.0.9beta7-10
1.0.9rc2	1	10009	2.1.0.9rc2
1.0.9	1	10009	2.1.0.9
1.0.10beta1	1	10010	2.1.0.10beta1
1.0.10rc1	1	10010	2.1.0.10rc1
1.0.10	1	10010	2.1.0.10
1.0.11beta1-3	1	10011	2.1.0.11beta1-3
1.0.11rc1	1	10011	2.1.0.11rc1
1.0.11	1	10011	2.1.0.11
1.0.12beta1-2	2	10012	2.1.0.12beta1-2
1.0.12rc1	2	10012	2.1.0.12rc1
1.0.12	2	10012	2.1.0.12
1.1.0a-f	-	10100	2.1.1.0a-f abandoned
1.2.0beta1-2	2	10200	2.1.2.0beta1-2
1.2.0beta3-5	3	10200	3.1.2.0beta3-5
1.2.0rc1	3	10200	3.1.2.0rc1
1.2.0	3	10200	3.1.2.0
1.2.1beta-4	3	10201	3.1.2.1beta1-4
1.2.1rc1-2	3	10201	3.1.2.1rc1-2
1.2.1	3	10201	3.1.2.1
1.2.2beta1-6	12	10202	12.so.0.1.2.2beta1-6
1.0.13beta1	10	10013	10.so.0.1.0.13beta1
1.0.13rc1	10	10013	10.so.0.1.0.13rc1
1.2.2rc1	12	10202	12.so.0.1.2.2rc1
1.0.13	10	10013	10.so.0.1.0.13
1.2.2	12	10202	12.so.0.1.2.2
1.2.3rc1-6	12	10203	12.so.0.1.2.3rc1-6
1.2.3	12	10203	12.so.0.1.2.3
1.2.4beta1-3	13	10204	12.so.0.1.2.4beta1-3
1.2.4rc1	13	10204	12.so.0.1.2.4rc1
1.0.14	10	10014	10.so.0.1.0.14
1.2.4	13	10204	12.so.0.1.2.4
1.2.5beta1-2	13	10205	12.so.0.1.2.5beta1-2
1.0.15rc1	10	10015	10.so.0.1.0.15rc1
1.0.15	10	10015	10.so.0.1.0.15
1.2.5	13	10205	12.so.0.1.2.5
1.2.6beta1-4	13	10206	12.so.0.1.2.6beta1-4
1.2.6rc1-5	13	10206	12.so.0.1.2.6rc1-5
1.0.16	10	10016	10.so.0.1.0.16
1.2.6	13	10206	12.so.0.1.2.6
1.2.7beta1-2	13	10207	12.so.0.1.2.7beta1-2
1.0.17rc1	10	10017	12.so.0.1.0.17rc1
1.2.7rc1	13	10207	12.so.0.1.2.7rc1
1.0.17	10	10017	12.so.0.1.0.17
1.2.7	13	10207	12.so.0.1.2.7
1.2.8beta1-5	13	10208	12.so.0.1.2.8beta1-5

1.0.18rc1-5	10	10018	12.so.0.1.0.18rc1-5
1.2.8rc1-5	13	10208	12.so.0.1.2.8rc1-5
1.0.18	10	10018	12.so.0.1.0.18
1.2.8	13	10208	12.so.0.1.2.8

Henceforth the source version will match the shared-library minor and patch numbers; the shared-library major version number will be used for changes in backward compatibility, as it is intended. The PNG_PNGLIB_VER macro, which is not used within libpng but is available for applications, is an unsigned integer of the form xyyzz corresponding to the source version x.y.z (leading zeros in y and z). Beta versions were given the previous public release number plus a letter, until version 1.0.6j; from then on they were given the upcoming public release number plus "betaNN" or "rcN".

SEE ALSO

libpngpf(3), png(5)

libpng:

http://libpng.sourceforge.net (follow the [DOWNLOAD] link) http://www.libpng.org/pub/png

zlib:

(generally) at the same location as *libpng* or at
ftp://ftp.info-zip.org/pub/infozip/zlib

PNG specification: *RFC* 2083

(generally) at the same location as *libpng* or at
ftp://ds.internic.net/rfc/rfc2083.txt
or (as a W3C Recommendation) at
http://www.w3.org/TR/REC-png.html

In the case of any inconsistency between the PNG specification and this library, the specification takes precedence.

AUTHORS

This man page: Glenn Randers-Pehrson <glennrp at users.sourceforge.net>

The contributing authors would like to thank all those who helped with testing, bug fixes, and patience. This wouldn't have been possible without all of you.

Thanks to Frank J. T. Wojcik for helping with the documentation.

Libpng version 1.2.8 - December 3, 2004: Initially created in 1995 by Guy Eric Schalnat, then of Group 42, Inc. Currently maintained by Glenn Randers-Pehrson (glennrp at users.sourceforge.net).

Supported by the PNG development group
png-implement at ccrc.wustl.edu (subscription required; write to majordomo at ccrc.wustl.edu with "subscribe png-implement" in the message).

COPYRIGHT NOTICE, DISCLAIMER, and LICENSE:

Cosmin Truta

libpng versions 1.0.7, July 1, 2000, through 1.2.5 - October 3, 2002, are Copyright (c) 2000-2002 Glenn Randers-Pehrson, and are distributed according to the same disclaimer and license as libpng-1.0.6 with the following individuals added to the list of Contributing Authors

Simon-Pierre Cadieux
Eric S. Raymond
Gilles Vollant

and with the following additions to the disclaimer:

There is no warranty against interference with your enjoyment of the library or against infringement. There is no warranty that our efforts or the library will fulfill any of your particular purposes or needs. This library is provided with all faults, and the entire risk of satisfactory quality, performance, accuracy, and effort is with the user.

libpng versions 0.97, January 1998, through 1.0.6, March 20, 2000, are Copyright (c) 1998, 1999 Glenn Randers-Pehrson Distributed according to the same disclaimer and license as libpng-0.96, with the following individuals added to the list of Contributing Authors:

Tom Lane
Glenn Randers-Pehrson
Willem van Schaik

libpng versions 0.89, June 1996, through 0.96, May 1997, are Copyright (c) 1996, 1997 Andreas Dilger Distributed according to the same disclaimer and license as libpng-0.88, with the following individuals added to the list of Contributing Authors:

John Bowler
Kevin Bracey
Sam Bushell
Magnus Holmgren
Greg Roelofs
Tom Tanner

libpng versions 0.5, May 1995, through 0.88, January 1996, are Copyright (c) 1995, 1996 Guy Eric Schalnat, Group 42, Inc.

For the purposes of this copyright and license, "Contributing Authors" is defined as the following set of individuals:

Andreas Dilger
Dave Martindale
Guy Eric Schalnat
Paul Schmidt
Tim Wegner

The PNG Reference Library is supplied "AS IS". The Contributing Authors and Group 42, Inc. disclaim all warranties, expressed or implied, including, without limitation, the warranties of merchantability and of fitness for any purpose. The Contributing Authors and Group 42, Inc. assume no liability for direct, indirect, incidental, special, exemplary, or consequential damages, which may result from the use of the PNG Reference Library, even if advised of the possibility of such damage.

Permission is hereby granted to use, copy, modify, and distribute this source code, or portions hereof, for any purpose, without fee, subject to the following restrictions:

1. The origin of this source code must not be misrepresented.

2. Altered versions must be plainly marked as such and
   must not be misrepresented as being the original source.

3. This Copyright notice may not be removed or altered from
   any source or altered source distribution.

The Contributing Authors and Group 42, Inc. specifically permit, without fee, and encourage the use of this source code as a component to supporting the PNG file format in commercial products. If you use this source code in a product, acknowledgment is not required but would be appreciated.

A "png_get_copyright" function is available, for convenient use in "about" boxes and the like:

```
printf("%s",png_get_copyright(NULL));
```

Also, the PNG logo (in PNG format, of course) is supplied in the files "pngbar.png" and "pngbar.jpg (88x31) and "pngnow.png" (98x31).

Libpng is OSI Certified Open Source Software. OSI Certified Open Source is a certification mark of the Open Source Initiative.

Glenn Randers-Pehrson glennrp at users.sourceforge.net December 3, 2004

NAME

libpng − Portable Network Graphics (PNG) Reference Library 1.2.8 (private functions)

SYNOPSIS

#include <png.h>

void png_build_gamma_table (png_structp *png_ptr***);**

void png_build_grayscale_palette (int *bit_depth***, png_colorp** *palette***);**

void png_calculate_crc (png_structp *png_ptr***, png_bytep** *ptr***, png_size_t** *length***);**

void png_check_chunk_name (png_structp *png_ptr***, png_bytep** *chunk_name***);**

png_size_t png_check_keyword (png_structp *png_ptr***, png_charp** *key***, png_charpp** *new_key***);**

void png_combine_row (png_structp *png_ptr***, png_bytep** *row***, int** *mask***);**

void png_correct_palette (png_structp *png_ptr*, **png_colorp** *palette*, **int** *num_palette*);

int png_crc_error (png_structp *png_ptr*);

int png_crc_finish (png_structp *png_ptr*, **png_uint_32** *skip*);

void png_crc_read (png_structp *png_ptr*, **png_bytep** *buf*, **png_size_t** *length*);

png_voidp png_create_struct (int *type*);

png_voidp png_create_struct_2 (int *type*, **png_malloc_ptr** *malloc_fn*, **png_voidp** *mem_ptr*);

png_charp png_decompress_chunk (png_structp *png_ptr*, **int** *comp_type*, **png_charp** *chunkdata*, **png_size_t** *chunklength*, **png_size_t** *prefix_length*, **png_size_t** **data_length*);

void png_destroy_struct (png_voidp *struct_ptr***);**

void png_destroy_struct_2 (png_voidp *struct_ptr***, png_free_ptr** *free_fn***, png_voidp** *mem_ptr***);**

void png_do_background (png_row_infop *row_info***, png_bytep** *row***, png_color_16p** *trans_values***, png_color_16p** *background***, png_color_16p** *background_1***, png_bytep** *gamma_table***, png_bytep** *gamma_from_1***, png_bytep** *gamma_to_1***, png_uint_16pp** *gamma_16***, png_uint_16pp** *gamma_16_from_1***, png_uint_16pp** *gamma_16_to_1***, int** *gamma_shift***);**

void png_do_bgr (png_row_infop *row_info***, png_bytep** *row***);**

void png_do_chop (png_row_infop *row_info***, png_bytep** *row***);**

void png_do_dither (png_row_infop *row_info***, png_bytep** *row***, png_bytep** *palette_lookup***, png_bytep** *dither_lookup***);**

void png_do_expand (png_row_infop *row_info***, png_bytep** *row***, png_color_16p** *trans_value***);**

void png_do_expand_palette (png_row_infop *row_info***, png_bytep** *row***, png_colorp** *palette***, png_bytep** *trans***, int** *num_trans***);**

void png_do_gamma (png_row_infop *row_info***, png_bytep** *row***, png_bytep** *gamma_table***, png_uint_16pp** *gamma_16_table***, int** *gamma_shift***);**

void png_do_gray_to_rgb (png_row_infop *row_info***, png_bytep** *row***);**

void png_do_invert (png_row_infop *row_info***, png_bytep** *row***);**

void png_do_pack (png_row_infop *row_info***, png_bytep** *row***, png_uint_32** *bit_depth***);**

void png_do_packswap (png_row_infop *row_info***, png_bytep** *row***);**

void png_do_read_filler (png_row_infop *row_info***, png_bytep** *row***, png_uint_32** *filler***, png_uint_32** *flags***);**

void png_do_read_interlace (png_row_infop *row_info*, **png_bytep** *row*, **int** *pass*, **png_uint_32** *transformations*);

void png_do_read_invert_alpha (png_row_infop *row_info*, **png_bytep** *row*);

void png_do_read_swap_alpha (png_row_infop *row_info*, **png_bytep** *row*);

void png_do_read_transformations (png_structp *png_ptr*);

int png_do_rgb_to_gray (png_row_infop *row_info*, **png_bytep** *row*);

void png_do_shift (png_row_infop *row_info*, **png_bytep** *row*, **png_color_8p** *bit_depth*);

void png_do_strip_filler (png_row_infop *row_info*, **png_bytep** *row*, **png_uint_32** *flags*);

void png_do_swap (png_row_infop *row_info*, **png_bytep** *row*);

void png_do_unpack (png_row_infop *row_info*, **png_bytep** *row*);

void png_do_unshift (png_row_infop *row_info*, **png_bytep** *row*, **png_color_8p** *sig_bits*);

void png_do_write_interlace (png_row_infop *row_info*, **png_bytep** *row*, **int** *pass*);

void png_do_write_invert_alpha (png_row_infop *row_info*, **png_bytep** *row*);

void png_do_write_swap_alpha (png_row_infop *row_info*, **png_bytep** *row*);

void png_do_write_transformations (png_structp *png_ptr*);

void *png_far_to_near (png_structp png_ptr,png_voidp *ptr*, **int** *check*);

void png_flush (png_structp *png_ptr***);**

png_int_32 png_get_int_32 (png_bytep *buf***);**

png_uint_16 png_get_uint_16 (png_bytep *buf***);**

png_uint_32 png_get_uint_31 (png_bytep *buf***);**

png_uint_32 png_get_uint_32 (png_bytep *buf***);**

void png_handle_bKGD (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *length***);**

void png_handle_cHRM (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *length***);**

void png_handle_gAMA (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *length***);**

void png_handle_hIST (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *length***);**

void png_handle_IEND (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *length***);**

void png_handle_IHDR (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *length***);**

void png_handle_iCCP (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *length***);**

void png_handle_iTXt (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *length***);**

void png_handle_oFFs (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *length***);**

void png_handle_pCAL (png_structp *png_ptr***, png_infop** *info_ptr***, png_uint_32** *length***);**

void png_handle_pHYs (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_handle_PLTE (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_handle_sBIT (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_handle_sCAL (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_handle_sPLT (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_handle_sRGB (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_handle_tEXt (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_handle_tIME (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_handle_tRNS (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_handle_unknown (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_handle_zTXt (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_info_destroy (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_init_mmx_flags (png_structp *png_ptr*);

void png_init_read_transformations (png_structp *png_ptr*);

void png_process_IDAT_data (png_structp *png_ptr*, **png_bytep** *buffer*, **png_size_t** *buffer_length*);

void png_process_some_data (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_push_check_crc (png_structp *png_ptr*);

void png_push_crc_finish (png_structp *png_ptr*);

void png_push_crc_skip (png_structp *png_ptr*, **png_uint_32** *length*);

void png_push_fill_buffer (png_structp *png_ptr*, **png_bytep** *buffer*, **png_size_t** *length*);

void png_push_handle_tEXt (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_push_handle_unknown (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_push_handle_zTXt (png_structp *png_ptr*, **png_infop** *info_ptr*, **png_uint_32** *length*);

void png_push_have_end (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_push_have_info (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_push_have_row (png_structp *png_ptr*, **png_bytep** *row*);

void png_push_process_row (png_structp *png_ptr*);

void png_push_read_chunk (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_push_read_end (png_structp *png_ptr*, **png_infop** *info_ptr*);

void png_push_read_IDAT (png_structp *png_ptr*);

void png_push_read_sig (png_structp *png_ptr***, png_infop** *info_ptr***);**

void png_push_read_tEXt (png_structp *png_ptr***, png_infop** *info_ptr***);**

void png_push_read_zTXt (png_structp *png_ptr***, png_infop** *info_ptr***);**

void png_push_restore_buffer (png_structp *png_ptr***, png_bytep** *buffer***, png_size_t** *buffer_length***);**

void png_push_save_buffer (png_structp *png_ptr***);**

void png_read_data (png_structp *png_ptr***, png_bytep** *data***, png_size_t** *length***);**

void png_read_filter_row (png_structp *png_ptr***, png_row_infop** *row_info***, png_bytep** *row***, png_bytep** *prev_row***, int** *filter***);**

void png_read_finish_row (png_structp *png_ptr***);**

void png_read_push_finish_row (png_structp *png_ptr***);**

void png_read_start_row (png_structp *png_ptr***);**

void png_read_transform_info (png_structp *png_ptr***, png_infop** *info_ptr***);**

void png_reset_crc (png_structp *png_ptr***);**

void png_save_int_32 (png_bytep *buf***, png_int_32** *i***);**

void png_save_uint_16 (png_bytep *buf***, unsigned int** *i***);**

void png_save_uint_32 (png_bytep *buf***, png_uint_32** *i***);**

int png_set_text_2 (png_structp *png_ptr***, png_infop** *info_ptr***, png_textp** *text_ptr***, int** *num_text***);**

void png_write_cHRM (png_structp *png_ptr***, double** *white_x***, double** *white_y***, double** *red_x***, double** *red_y***, double** *green_x***, double** *green_y***, double** *blue_x***, double** *blue_y***);**

void png_write_cHRM_fixed (png_structp *png_ptr***, png_uint_32** *white_x***, png_uint_32** *white_y***, png_uint_32** *red_x***, png_uint_32** *red_y***, png_uint_32** *green_x***, png_uint_32** *green_y***, png_uint_32** *blue_x***, png_uint_32** *blue_y***);**

void png_write_data (png_structp *png_ptr***, png_bytep** *data***, png_size_t** *length***);**

void png_write_filtered_row (png_structp *png_ptr***, png_bytep** *filtered_row***);**

void png_write_find_filter (png_structp *png_ptr***, png_row_infop** *row_info***);**

void png_write_finish_row (png_structp *png_ptr***);**

void png_write_gAMA (png_structp *png_ptr***, double** *file_gamma***);**

void png_write_gAMA_fixed (png_structp *png_ptr***, png_uint_32** *int_file_gamma***);**

void png_write_hIST (png_structp *png_ptr***, png_uint_16p** *hist***, int** *num_hist***);**

void png_write_iCCP (png_structp *png_ptr***, png_charp** *name***, int** *compression_type***, png_charp** *profile***, int** *proflen***);**

void png_write_IDAT (png_structp *png_ptr***, png_bytep** *data***, png_size_t** *length***);**

void png_write_IEND (png_structp *png_ptr***);**

void png_write_IHDR (png_structp *png_ptr***, png_uint_32** *width***, png_uint_32** *height***, int** *bit_depth***, int** *color_type***, int** *compression_type***, int** *filter_type***, int** *interlace_type***);**

void png_write_iTXt (png_structp *png_ptr*, **int** *compression*, **png_charp** *key*, **png_charp** *lang*, **png_charp** *translated_key*, **png_charp** *text*);

void png_write_oFFs (png_structp *png_ptr*, **png_uint_32** *x_offset*, **png_uint_32** *y_offset*, **int** *unit_type*);

void png_write_pCAL (png_structp *png_ptr*, **png_charp** *purpose*, **png_int_32** *X0*, **png_int_32** *X1*, **int** *type*, **int** *nparams*, **png_charp** *units*, **png_charpp** *params*);

void png_write_pHYs (png_structp *png_ptr*, **png_uint_32** *x_pixels_per_unit*, **png_uint_32** *y_pixels_per_unit*, **int** *unit_type*);

void png_write_PLTE (png_structp *png_ptr*, **png_colorp** *palette*, **png_uint_32** *num_pal*);

void png_write_sBIT (png_structp *png_ptr*, **png_color_8p** *sbit*, **int** *color_type*);

void png_write_sCAL (png_structp *png_ptr*, **png_charp** *unit*, **double** *width*, **double** *height*);

void png_write_sCAL_s (png_structp *png_ptr***, png_charp** *unit***, png_charp** *width***, png_charp** *height***);**

void png_write_sig (png_structp *png_ptr***);**

void png_write_sRGB (png_structp *png_ptr***, int** *intent***);**

void png_write_sPLT (png_structp *png_ptr***, png_spalette_p** *palette***);**

void png_write_start_row (png_structp *png_ptr***);**

void png_write_tEXt (png_structp *png_ptr***, png_charp** *key***, png_charp** *text***, png_size_t** *text_len***);**

void png_write_tIME (png_structp *png_ptr***, png_timep** *mod_time***);**

void png_write_tRNS (png_structp *png_ptr***, png_bytep** *trans***, png_color_16p** *values***, int** *number***, int** *color_type***);**

void png_write_zTXt (png_structp *png_ptr***, png_charp** *key***, png_charp** *text***, png_size_t** *text_len***, int** *compression***);**

voidpf png_zalloc (voidpf *png_ptr***, uInt** *items***, uInt** *size***);**

void png_zfree (voidpf *png_ptr***, voidpf** *ptr***);**

DESCRIPTION

The functions listed above are used privately by libpng and are not recommended for use by applications. They are not "exported" to applications using shared libraries. They are listed alphabetically here as an aid to libpng maintainers. See png.h for more information on these functions.

SEE ALSO

libpng(3), png(5)

AUTHOR

Glenn Randers-Pehrson

NAME

png − Portable Network Graphics (PNG) format

DESCRIPTION

PNG (Portable Network Graphics) is an extensible file format for the lossless, portable, well-compressed storage of raster images. PNG provides a patent-free replacement for GIF and can also replace many common uses of TIFF. Indexed-color, grayscale, and truecolor images are supported, plus an optional alpha channel. Sample depths range from 1 to 16 bits.

PNG is designed to work well in online viewing applications, such as the World Wide Web, so it is fully streamable with a progressive display option. PNG is robust, providing both full file integrity checking and fast, simple detection of common transmission errors. Also, PNG can store gamma and chromaticity data for improved color matching on heterogeneous platforms.

SEE ALSO

libpng(3),zlib(3),*deflate(5)*,and*zlib(5)*

PNG specification (second edition), November 2003:

<http://www.w3.org/TR/2003/REC-PNG-20031110/ PNG 1.2 specification, July 1999:

http://www.libpng.org/pub/png

PNG 1.0 specification, October 1996:

RFC 2083

ftp://ds.internic.net/rfc/rfc2083.txt
or (as a W3C Recommendation) at
http://www.w3.org/TR/REC-png.html

AUTHORS

This man page: Glenn Randers-Pehrson

Portable Network Graphics (PNG) Specification (Second Edition) Information technology - Computer graphics and image processing - Portable Network Graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E) (November 10, 2003): David Duce and others.

Portable Network Graphics (PNG) Specification Version 1.2 (July 8, 1999): Glenn Randers-Pehrson and others (png-list).

Portable Network Graphics (PNG) Specification Version 1.0 (October 1, 1996): Thomas Boutell and others (png-list).

COPYRIGHT NOTICE