1.

```
#define zero_t ( ( math_data ∗ ) mp→math ) → zero_t
#define number_zero(A) ( ( ( math_data ∗ ) (mp→math) ) → equal ) (A, zero_t)
#define number_greater(A, B) ( ( ( math_data ∗ ) (mp→math) ) → greater ) (A, B)
#define number_positive(A) number_greater(A, zero_t)
#define number_to_scaled(A) ( ( ( math_data ∗ ) (mp→math) ) → to_scaled ) (A)
#define round_unscaled(A) ( ( ( math_data ∗ ) (mp→math) ) → round_unscaled ) (A)
#define true 1
#define false 0
#define null_font 0
#define null 0
#define unity 1.0
#define incr(A) (A) = (A) + 1    /∗ increase a variable by unity ∗/
#define decr(A) (A) = (A) − 1    /∗ decrease a variable by unity ∗/
#define negate(A) (A) = −(A)     /∗ change the sign of a variable ∗/
#include <w2c/config.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "mplib.h"
#include "mplibps.h"    /∗ external header ∗/
#include "mplibpng.h"    /∗ external header ∗/
#include "mpmp.h"    /∗ internal header ∗/
#include "mppsout.h"    /∗ internal header ∗/
#include "mppngout.h"    /∗ internal header ∗/
#include "mpmath.h"    /∗ internal header ∗/
  ⟨ Preprocessor definitions ⟩
  ⟨ Types in the outer block 12 ⟩⟨ Declarations 8 ⟩
```

2. There is a small bit of code from the backend that bleads through to the frontend because I do not know how to set up the includes properly. That is **typedef struct** *pngout_data_struct* ∗**pngout_data**.

3. ⟨ mppngout.h 3 ⟩ ≡

```
#ifndef MPPNGOUT_H
#define MPPNGOUT_H 1
#include "cairo.h"
#define PNG_SKIP_SETJMP_CHECK 1
#include "png.h"
#include "mplib.h"
#include "mpmp.h"
#include "mplibps.h"
  typedef struct pngout_data_struct {
    ⟨ Globals 6 ⟩
  } pngout_data_struct; ⟨ Exported function headers 4 ⟩
#endif
```

4. ⟨ Exported function headers 4 ⟩ ≡

```
  void mp_png_backend_initialize(MP mp);
  void mp_png_backend_free(MP mp);
```

See also section 30.

This code is used in section 3.

5.

```
void mp_png_backend_initialize(MP mp)
{
  mp→png = mp_xmalloc(mp, 1, sizeof(pngout_data_struct));
  memset(mp→png, 0, sizeof(pngout_data_struct));
}
void mp_png_backend_free(MP mp)
{
  mp_xfree(mp→png);
  mp→png = Λ;
}
```

6. Writing to PNG files

⟨ Globals 6 ⟩ ≡

```
  cairo_surface_t ∗ surface;
  cairo_t ∗ cr;
```

See also sections 7 and 23.

This code is used in section 3.

7. We often need to print a pair of coordinates, and these need to offset so that all coordinates are positive.

⟨ Globals 6 ⟩ +≡

```
  integer dx;
  integer dy;
```

8.

⟨ Declarations 8 ⟩ ≡

```
  static void mp_png_start(MP mp, mp_edge_object ∗ hh, double hppp, double vppp, int colormodel, int
        antialias);
```

See also sections 11, 13, 16, 19, 22, 24, 26, 28, and 34.

This code is used in section 1.

9.

```
void mp_png_start(MP mp, mp_edge_object *hh, double hppp, double vppp, int colormodel, int antialias)
{
  double w, h;
  if (hh→minx > hh→maxx) {
    w = 1;
    h = 1;
    mp→png→dx = 0;
    mp→png→dy = 0;
  }
  else {
    w = (ceil(hh→maxx) − floor(hh→minx))/hppp;
    h = (ceil(hh→maxy) − floor(hh→miny))/vppp;
    mp→png→dx = −floor(hh→minx);
    mp→png→dy = −floor(hh→miny);
  }
  mp→png→surface = cairo_image_surface_create(CAIRO_FORMAT_ARGB32, w, h);
  mp→png→cr = cairo_create(mp→png→surface);
    /* if there is no alpha channel, a white background is needed */
  if (colormodel ≡ PNG_COLOR_TYPE_RGB ∨ colormodel ≡ PNG_COLOR_TYPE_GRAY) {
    cairo_save(mp→png→cr);
    cairo_set_source_rgb(mp→png→cr, 1.0, 1.0, 1.0);
    cairo_rectangle(mp→png→cr, 0, 0, w, h);
    cairo_fill(mp→png→cr);
    cairo_restore(mp→png→cr);
  }
  cairo_scale(mp→png→cr, 1/hppp, −1/vppp);
  cairo_translate(mp→png→cr, 0, −(h * vppp));
  cairo_translate(mp→png→cr, mp→png→dx, mp→png→dy);
  cairo_set_antialias(mp→png→cr, antialias);
}
```

10. Outputting a color specification.

```
#define set_color_objects(pq) object_color_model = pq→color_model;
            object_color_a = pq→color.a_val;
            object_color_b = pq→color.b_val;
            object_color_c = pq→color.c_val;
            object_color_d = pq→color.d_val;

  static void mp_png_color_out(MP mp, mp_graphic_object *p){ int object_color_model;
        double object_color_a, object_color_b, object_color_c, object_color_d; if (gr_type(p) ≡ mp_fill_code) {
            mp_fill_object *pq = ( mp_fill_object * ) p;
        set_color_objects(pq); } else if (gr_type(p) ≡ mp_stroked_code) { mp_stroked_object *pq = (
            mp_stroked_object * ) p;
        set_color_objects(pq); } else { mp_text_object *pq = ( mp_text_object * ) p;
        set_color_objects(pq); }
        if (object_color_model ≡ mp_no_model) {
          cairo_set_source_rgb(mp→png→cr, 0, 0, 0);
        }
        else {
          if (object_color_model ≡ mp_grey_model) {
            object_color_b = object_color_a;
            object_color_c = object_color_a;
          }
          else if (object_color_model ≡ mp_cmyk_model) {
            int c, m, y, k;

            c = object_color_a;
            m = object_color_b;
            y = object_color_c;
            k = object_color_d;
            object_color_a = unity − (c + k > unity ? unity : c + k);
            object_color_b = unity − (m + k > unity ? unity : m + k);
            object_color_c = unity − (y + k > unity ? unity : y + k);
          }
          cairo_set_source_rgb(mp→png→cr, object_color_a, object_color_b, object_color_c);
        }
        }
```

11. ⟨ Declarations 8 ⟩ +≡

```
  static void mp_png_color_out(MP mp, mp_graphic_object *p);
```

12. This is the information that comes from a pen

⟨ Types in the outer block 12 ⟩ ≡

```
  typedef struct mp_pen_info {
    double tx, ty;
    double sx, rx, ry, sy;
    double ww;
  } mp_pen_info;
```

See also sections 31 and 32.

This code is used in section 1.

13. (Re)discover the characteristics of an elliptical pen

⟨ Declarations 8 ⟩ +≡

```
  mp_pen_info *mp_png_pen_info(MP mp, mp_gr_knot pp, mp_gr_knot p);
```

14. The next two constants come from the original web source. Together with the two helper functions, they will tell whether the x or the y direction of the path is the most important

```
#define aspect_bound (10/65536.0)
#define aspect_default 1
  static double coord_range_x(mp_gr_knot h, double dz)
  {
    double z;
    double zlo = 0, zhi = 0;
    mp_gr_knot f = h;
    while (h ≠ Λ) {
      z = gr_x_coord(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      z = gr_right_x(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      z = gr_left_x(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      h = gr_next_knot(h);
      if (h ≡ f) break;
    }
    return (zhi − zlo ≤ dz ? aspect_bound : aspect_default);
  }
  static double coord_range_y(mp_gr_knot h, double dz)
  {
    double z;
    double zlo = 0, zhi = 0;
    mp_gr_knot f = h;
    while (h ≠ Λ) {
      z = gr_y_coord(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      z = gr_right_y(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      z = gr_left_y(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      h = gr_next_knot(h);
      if (h ≡ f) break;
    }
    return (zhi − zlo ≤ dz ? aspect_bound : aspect_default);
  }
```

15.

```
mp_pen_info *mp_png_pen_info(MP mp, mp_gr_knot pp, mp_gr_knot p)
{
  double wx, wy;    /* temporary pen widths, in either direction */
  struct mp_pen_info *pen;    /* return structure */
  if (p ≡ Λ) return Λ;
  pen = mp_xmalloc(mp, 1, sizeof(mp_pen_info));
  pen→rx = unity;
  pen→ry = unity;
  pen→ww = unity;
  if ((gr_right_x(p) ≡ gr_x_coord(p)) ∧ (gr_left_y(p) ≡ gr_y_coord(p))) {
    wx = fabs(gr_left_x(p) − gr_x_coord(p));
    wy = fabs(gr_right_y(p) − gr_y_coord(p));
  }
  else {
    double arg1, arg2;
    arg1 = gr_left_x(p) − gr_x_coord(p);
    arg2 = gr_right_x(p) − gr_x_coord(p);
    wx = sqrt(arg1 * arg1 + arg2 * arg2);
    arg1 = gr_left_y(p) − gr_y_coord(p);
    arg2 = gr_right_y(p) − gr_y_coord(p);
    wy = sqrt(arg1 * arg1 + arg2 * arg2);
  }
  if ((wy/coord_range_x(pp, wx)) ≥ (wx/coord_range_y(pp, wy))) pen→ww = wy;
  else pen→ww = wx;
  pen→tx = gr_x_coord(p);
  pen→ty = gr_y_coord(p);
  pen→sx = gr_left_x(p) − pen→tx;
  pen→rx = gr_left_y(p) − pen→ty;
  pen→ry = gr_right_x(p) − pen→tx;
  pen→sy = gr_right_y(p) − pen→ty;
  if (pen→ww ≠ unity) {
    if (pen→ww ≡ 0) {
      pen→sx = unity;
      pen→sy = unity;
    }
    else {
      pen→rx = pen→rx/pen→ww;
      pen→ry = pen→ry/pen→ww;
      pen→sx = pen→sx/pen→ww;
      pen→sy = pen→sy/pen→ww;
    }
  }
  return pen;
}
```

16. Two types of straight lines come up often in METAPOST paths: cubics with zero initial and final velocity as created by *make_path* or *make_envelope*, and cubics with control points uniformly spaced on a line as created by *make_choices*.

⟨ Declarations 8 ⟩ +≡

```
  static boolean mp_is_curved(mp_gr_knot p, mp_gr_knot q);
```

17.

```
#define bend_tolerance (131/65536.0)    /* allow rounding error of $2 \cdot 10^{-3}$ */
  boolean mp_is_curved(mp_gr_knot p, mp_gr_knot q)
  {
    double d;    /* a temporary value */
    if (gr_right_x(p) ≡ gr_x_coord(p))
      if (gr_right_y(p) ≡ gr_y_coord(p))
        if (gr_left_x(q) ≡ gr_x_coord(q))
          if (gr_left_y(q) ≡ gr_y_coord(q)) return false;
    d = gr_left_x(q) − gr_right_x(p);
    if (fabs(gr_right_x(p) − gr_x_coord(p) − d) ≤ bend_tolerance)
      if (fabs(gr_x_coord(q) − gr_left_x(q) − d) ≤ bend_tolerance) {
        d = gr_left_y(q) − gr_right_y(p);
        if (fabs(gr_right_y(p) − gr_y_coord(p) − d) ≤ bend_tolerance)
          if (fabs(gr_y_coord(q) − gr_left_y(q) − d) ≤ bend_tolerance) return false;
      }
    return true;
  }
```

18. Cairo does not want to draw a path that consists of only a moveto, so make sure there is some kind of line even for single-pair paths.

```
  static void mp_png_path_out(MP mp, mp_gr_knot h)
  {
    mp_gr_knot p, q;    /* for scanning the path */
    int steps = 0;
    cairo_move_to(mp→png→cr, gr_x_coord(h), gr_y_coord(h));
    p = h;
    do {
      if (gr_right_type(p) ≡ mp_endpoint) {
        if (steps ≡ 0) {
          cairo_line_to(mp→png→cr, gr_x_coord(p), gr_y_coord(p));
        }
        return;
      }
      q = gr_next_knot(p);
      if (mp_is_curved(p, q)) {
        cairo_curve_to(mp→png→cr, gr_right_x(p), gr_right_y(p), gr_left_x(q), gr_left_y(q), gr_x_coord(q),
            gr_y_coord(q));
      }
      else {
        cairo_line_to(mp→png→cr, gr_x_coord(q), gr_y_coord(q));
      }
      p = q;
      steps++;
    } while (p ≠ h);
    if ((gr_x_coord(p) ≡ gr_x_coord(h)) ∧ (gr_y_coord(p) ≡ gr_y_coord(h)) ∧ gr_right_type(p) ≠ mp_endpoint)
      {
      cairo_close_path(mp→png→cr);
    }
  }
```

19. Now for outputting the actual graphic objects.

```
⟨ Declarations 8 ⟩ +≡
  static double mp_png_choose_scale(MP mp, mp_graphic_object *p);
```

20.

```
double mp_png_choose_scale(MP mp, mp_graphic_object *p)
  {    /* p should point to a text node */
    double a, b, c, d, ad, bc;    /* temporary values */
    double ret1, ret2;

    a = gr_txx_val(p);
    b = gr_txy_val(p);
    c = gr_tyx_val(p);
    d = gr_tyy_val(p);
    if (a < 0) negate(a);
    if (b < 0) negate(b);
    if (c < 0) negate(c);
    if (d < 0) negate(d);
    ad = (a - d)/2.0;
    bc = (b - c)/2.0;
    ret1 = sqrt((d + ad) * (d + ad) + ad * ad);
    ret2 = sqrt((c + bc) * (c + bc) + bc * bc);
    return sqrt(ret1 * ret1 + ret2 * ret2);
  }
```

21.

```
#define xrealloc(P, A, B) mp_xrealloc(mp, P, (size_t) A, B)
#define XREALLOC(a, b, c) a = xrealloc(a, (b + 1), sizeof (c));

  void mp_reallocate_psfonts(MP mp, int l){ if (l ≥ mp→png→font_max) { int f;

      mp→png→last_fnum = mp→png→font_max; XREALLOC (mp→png→psfonts, l, mp_edge_object * ) ;
      for (f = (mp→png→last_fnum + 1); f ≤ l; f++) {
        mp→png→psfonts[f] = Λ;
      }
      mp→png→font_max = l; } }
```

22.

```
⟨ Declarations 8 ⟩ +≡
  void mp_reallocate_psfonts(MP mp, int l);
```

23.

```
⟨ Globals 6 ⟩ +≡
  mp_edge_object **psfonts;
  int font_max;
  int last_fnum;
```

24.

```
⟨ Declarations 8 ⟩ +≡
  static void mp_png_text_out(MP mp, mp_text_object *p);
```

25.
```
void mp_png_text_out(MP mp, mp_text_object ∗ p){ double ds;
    /∗ design size and scale factor for a text node ∗/
  unsigned char ∗s = (unsigned char ∗) gr_text_p(p);
  size_t l = gr_text_l(p);     /∗ string length ∗/
  boolean transformed = (gr_txx_val(p) ≠ unity) ∨ (gr_tyy_val(p) ≠ unity) ∨ (gr_txy_val(p) ≠
      0) ∨ (gr_tyx_val(p) ≠ 0);
  int fn = gr_font_n(p);
  mp_ps_font ∗ f;
  double scf;
  ds = (mp→font_dsize[fn] + 8)/(16 ∗ 65536.0); scf = mp_png_choose_scale (mp, ( mp_graphic_object ∗ )
      p ) ;
  cairo_save(mp→png→cr);
  if (transformed) {
    cairo_matrix_t matrix = {0, 0, 0, 0, 0, 0};
    cairo_matrix_init(&matrix, (gr_txx_val(p)/scf), (gr_tyx_val(p)/scf), (gr_txy_val(p)/scf),
        (gr_tyy_val(p)/scf), gr_tx_val(p), gr_ty_val(p));
    cairo_transform(mp→png→cr, &matrix);
    cairo_move_to(mp→png→cr, 0, 0);
  }
  else {
    cairo_translate(mp→png→cr, gr_tx_val(p), gr_ty_val(p));
  }
  cairo_scale(mp→png→cr, ((ds/1000.0) ∗ scf), ((ds/1000.0) ∗ scf)); mp_png_color_out (mp, (
      mp_graphic_object ∗ ) p ) ; while (l−− > 0) { mp_edge_object ∗ ch;
  int k = (int) ∗s++;
  double wd = 0.0;     /∗ this is in PS design units ∗/
  mp_reallocate_psfonts(mp, ((fn + 1) ∗ 256));
  ch = mp→png→psfonts[(fn ∗ 256) + k];
  if (ch ≡ Λ) {
    f = mp_ps_font_parse(mp, fn);
    if (f ≡ Λ) return;
    ch = mp_ps_font_charstring(mp, f, k);
    mp→png→psfonts[(fn ∗ 256) + k] = ch;
  }
  if (ch ≠ Λ) { mp_graphic_object ∗ pp = ch→body; while (pp ≠ Λ) { mp_png_path_out (mp, gr_path_p
      ( ( mp_fill_object ∗ ) pp ) ) ;
  pp = pp→next; } cairo_fill(mp→png→cr); } wd = mp_get_char_dimension(mp, mp→font_name[fn], k, 'w');

    /∗ wd/100 is the size in PS point , i.e, wd = 100 · real_wd but without considering scaling. We have
      a scale factor of (ds/1000.0) · scf so to match the scale wd should be 1000 · real_w d · scf/(ds · scf)
      i.e. wd = 10 · wd/ds. ∗/
  wd ∗= 10.0/ds;
  cairo_translate(mp→png→cr, wd, 0); } cairo_restore(mp→png→cr); }
```

26. When stroking a path with an elliptical pen, it is necessary to transform the coordinate system so that a unit circular pen will have the desired shape. To keep this transformation local, we enclose it in a

$$\langle g\rangle \ldots \langle /g\rangle$$

block. Any translation component must be applied to the path being stroked while the rest of the transformation must apply only to the pen. If *fill_also* = *true*, the path is to be filled as well as stroked so we must insert commands to do this after giving the path.

⟨ Declarations 8 ⟩ +≡
  static void *mp_png_stroke_out*(MP *mp*, *mp_graphic_object* ∗ *h*, **mp_pen_info** ∗*pen*, *boolean fill_also*);

27.
```
void mp_png_stroke_out(MP mp, mp_graphic_object * h, mp_pen_info *pen, boolean fill_also){
      boolean transformed = false; if (fill_also) { cairo_save(mp→png→cr); mp_png_path_out (mp,
      gr_path_p ( ( mp_stroked_object * ) h ) ) ;
  cairo_close_path(mp→png→cr);
  cairo_fill(mp→png→cr);
  cairo_restore(mp→png→cr); } cairo_save(mp→png→cr);
  if (pen ≠ Λ) {
    transformed = true;
    if ((pen→sx ≡ unity) ∧ (pen→rx ≡ 0) ∧ (pen→ry ≡ 0) ∧ (pen→sy ≡ unity) ∧ (pen→tx ≡ 0) ∧ (pen→ty ≡ 0))
      {
      transformed = false;
    }
  }
  if (pen ≠ Λ) {
    cairo_set_line_width(mp→png→cr, pen→ww);
  }
  else {
    cairo_set_line_width(mp→png→cr, 0);
  }
  if (gr_lcap_val(h) ≠ 0) {
    switch (gr_lcap_val(h)) {
    case 1: cairo_set_line_cap(mp→png→cr, CAIRO_LINE_CAP_ROUND);
      break;
    case 2: cairo_set_line_cap(mp→png→cr, CAIRO_LINE_CAP_SQUARE);
      break;
    default: cairo_set_line_cap(mp→png→cr, CAIRO_LINE_CAP_BUTT);
      break;
    }
  }
  if (gr_type(h) ≠ mp_fill_code) {
    mp_dash_object * hh = gr_dash_p(h);
    if (hh ≠ Λ ∧ hh→array ≠ Λ) {
      int i;
      for (i = 0; *(hh→array + i) ≠ −1; i++) ;
      cairo_set_dash(mp→png→cr, hh→array, i, hh→offset);
    }
  }
  if ( gr_ljoin_val ( ( mp_stroked_object * ) h ) ≠ 0 ) { switch ( gr_ljoin_val ( ( mp_stroked_object * ) h
      ) )
  {
  case 1: cairo_set_line_join(mp→png→cr, CAIRO_LINE_JOIN_ROUND);
    break;
  case 2: cairo_set_line_join(mp→png→cr, CAIRO_LINE_JOIN_BEVEL);
    break;
  default: cairo_set_line_join(mp→png→cr, CAIRO_LINE_JOIN_MITER);
    break;
  }
  } cairo_set_miter_limit (mp→png→cr, gr_miterlim_val ( ( mp_stroked_object * ) h ) ) ; mp_png_path_out
      (mp, gr_path_p ( ( mp_stroked_object * ) h ) ) ;
  if (transformed) {
    cairo_matrix_t matrix = {0, 0, 0, 0, 0, 0};
    cairo_save(mp→png→cr);
    cairo_matrix_init(&matrix, pen→sx, pen→rx, pen→ry, pen→sy, pen→tx, pen→ty);
    cairo_transform(mp→png→cr, &matrix);
    cairo_stroke(mp→png→cr);
    cairo_restore(mp→png→cr);
  }
  else {
    cairo_stroke(mp→png→cr);
  }
  cairo_restore(mp→png→cr); }
```

28. Here is a simple routine that just fills a cycle.

```
⟨ Declarations 8 ⟩ +≡
  static void mp_png_fill_out(MP mp, mp_gr_knot p, mp_graphic_object * h);
```

29.

```
void mp_png_fill_out(MP mp, mp_gr_knot p, mp_graphic_object * h)
  {
    cairo_save(mp→png→cr);
    mp_png_path_out(mp, p);
    cairo_close_path(mp→png→cr);
    cairo_fill(mp→png→cr);
    cairo_restore(mp→png→cr);
  }
```

30. The main output function

```
#define pen_is_elliptical(A)  ((A) ≡ gr_next_knot((A)))
#define gr_has_color(A)  (gr_type((A)) < mp_start_clip_code)
⟨ Exported function headers 4 ⟩ +≡
  int mp_png_gr_ship_out(mp_edge_object * hh, const char *options, int standalone);
```

31. This is a structure to ship data from cairo to our png writer. *width* and *height* could have been stored in our private *mp* instance, but this is just as easy.

```
⟨ Types in the outer block 12 ⟩ +≡
  typedef struct {
    unsigned char *data;
    int height;
    int width;
  } bitmap_t;
```

32. This is a small structure that is needed so that the png writer callbacks can actually access the *mp* object instance.

```
⟨ Types in the outer block 12 ⟩ +≡
  typedef struct {
    void *fp;
    MP mp;
  } mp_png_io;
```

33. Output a png chunk: the libpng callbacks

```
static void mp_write_png_data(png_structp png_ptr, png_bytep data, png_size_t length)
{
  mp_png_io *ioptr = (mp_png_io *) png_get_io_ptr(png_ptr);
  MP mp = ioptr→mp;
  (mp→write_binary_file)(mp, ioptr→fp, (void *) data, (size_t) length);
}
static void mp_write_png_flush(png_structp png_ptr)
{   /* nothing to do */
}
```

34. Write *bitmap* to a PNG file specified by *path*; returns 0 on success, non-zero on error. The original of this function was borrowed from an internet post, and extended as needed.

```
⟨ Declarations 8 ⟩ +≡
  int mp_png_save_to_file(MP mp, const bitmap_t *bitmap, const char *path, int colormodel);
```

35.
```
int mp_png_save_to_file(MP mp, const bitmap_t *bitmap, const char *path, int colormodel){
      mp_png_io io;
  png_structp png_ptr = Λ;
  png_infop info_ptr = Λ;
  size_t y;
  png_byte * *row_pointers = Λ;
  int status = −1;
  int depth = 8;
  int dpi = 72;
  int ppm_x;
  int ppm_y;      /* pixels per metre */
  io.mp = mp;
  io.fp = (mp→open_file)(mp, path, "wb", mp_filetype_bitmap);
  if (¬io.fp) {
    goto fopen_failed;
  }
  png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, Λ, Λ, Λ);
  if (png_ptr ≡ Λ) {
    goto png_create_write_struct_failed;
  }
  info_ptr = png_create_info_struct(png_ptr);
  if (info_ptr ≡ Λ) {
    goto png_create_info_struct_failed;
  }      /* Set up error handling. */
  if (setjmp(png_jmpbuf(png_ptr))) {
    goto png_failure;
  }      /* Set image attributes. */
  png_set_IHDR(png_ptr, info_ptr, bitmap→width, bitmap→height, depth, colormodel,
      PNG_INTERLACE_NONE, PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);
    /* Compression level 3 appears the best tradeoff between disk size and compression speed */
  png_set_compression_level(png_ptr, 3);
  png_set_filter(png_ptr, 0, PNG_FILTER_NONE);      /* setup some information */
  if (1) {
    png_text text[2];
    char *a, *b, *c, *d;      /* to get rid of a typecast warning */
    a = xstrdup("Title");
    b = xstrdup(path);
    c = xstrdup("Software");
    d = xstrdup("Generated␣by␣Metapost␣version␣" metapost_version);
    text[0].compression = PNG_TEXT_COMPRESSION_NONE;
    text[0].key = a;
    text[0].text = b;
    text[1].compression = PNG_TEXT_COMPRESSION_NONE;
    text[1].key = c;
    text[1].text = d;
    png_set_text(png_ptr, info_ptr, text, 2);
    free(a);
    free(b);
    free(c);
    free(d);
  }    /* The original plan was to add hppp and vppp values in here, but that seems to have negative
         effects on various bits of software. Better keep the DPI at 72 */
  ppm_x = dpi/0.0254;
  ppm_y = dpi/0.0254;
  png_set_pHYs(png_ptr, info_ptr, ppm_x, ppm_y, PNG_RESOLUTION_METER);
    /* Initialize rows of PNG. */
  row_pointers = malloc ( bitmap→height ∗ sizeof ( png_byte ∗ ) ) ;
  for (y = 0; y < bitmap→height; ++y) {
    if (colormodel ≡ PNG_COLOR_TYPE_GRAY) {
      row_pointers[y] = bitmap→data + bitmap→width ∗ y;
    }
    else if (colormodel ≡ PNG_COLOR_TYPE_GRAY_ALPHA) {
      row_pointers[y] = bitmap→data + bitmap→width ∗ 2 ∗ y;
    }
    else {
      row_pointers[y] = bitmap→data + bitmap→width ∗ 4 ∗ y;
    }
  }    /* Write the image data to io */
  png_set_write_fn(png_ptr, &io, mp_write_png_data, mp_write_png_flush);
  png_set_rows(png_ptr, info_ptr, row_pointers);
  if (colormodel ≡ PNG_COLOR_TYPE_RGB) {    /* Unfortunately, png_write_png does not have
         enough PNG_TRANSFORM options to do this properly, so we have to modify the bitmap data */
    int i;

    for (i = 0; i < bitmap→width ∗ bitmap→height ∗ 4; i += 4) {
      unsigned char b = bitmap→data[i];
      unsigned char g = bitmap→data[i + 1];

      bitmap→data[i] = bitmap→data[i + 3];
      bitmap→data[i + 1] = bitmap→data[i + 2];
      bitmap→data[i + 2] = g;
      bitmap→data[i + 3] = b;
    }
    png_write_png(png_ptr, info_ptr, PNG_TRANSFORM_STRIP_FILLER, Λ);
  }
  else if (colormodel ≡ PNG_COLOR_TYPE_RGB_ALPHA) {
    png_write_png(png_ptr, info_ptr, PNG_TRANSFORM_BGR, Λ);
  }
  else if (colormodel ≡ PNG_COLOR_TYPE_GRAY ∨ colormodel ≡ PNG_COLOR_TYPE_GRAY_ALPHA) {
    int i, j;

    j = 0;
    for (i = 0; i < bitmap→width ∗ bitmap→height ∗ 4; i += 4) {
      unsigned char b = bitmap→data[i];
      unsigned char g = bitmap→data[i + 1];
      unsigned char r = bitmap→data[i + 2];

      bitmap→data[j++] = ((r ≡ g ∧ r ≡ b) ? r : 0.2126 ∗ r + 0.7152 ∗ g + 0.0722 ∗ b);
      if (colormodel ≡ PNG_COLOR_TYPE_GRAY_ALPHA) bitmap→data[j++] = bitmap→data[i + 3];
    }
    png_write_png(png_ptr, info_ptr, PNG_TRANSFORM_IDENTITY, Λ);
  }
  status = 0;
  free(row_pointers);
png_failure: png_create_info_struct_failed: png_destroy_write_struct(&png_ptr, &info_ptr);
```

png_create_write_struct_failed: $(mp\neg close\_file)(mp, io.fp)$;
fopen_failed: **return** *status*; }

36.

```
#define number_to_double(A) ( ( ( math_data * ) (mp→math) ) → to_double ) (A)
  int mp_png_gr_ship_out(mp_edge_object * hh, const char *options, int standalone){ char *ss;
      mp_graphic_object * p;
      mp_pen_info *pen = Λ;
      MP mp = hh→parent;
      bitmap_t bitmap;
      const char *currentoption = options;
      int colormodel = PNG_COLOR_TYPE_RGB_ALPHA;
      int antialias = CAIRO_ANTIALIAS_FAST;
      int c;
      while (currentoption ∧ *currentoption) {
        if (strncmp(currentoption, "format=", 7) ≡ 0) {
          currentoption += 7;
          if (strncmp(currentoption, "rgba", 4) ≡ 0) {
            colormodel = PNG_COLOR_TYPE_RGB_ALPHA;
            currentoption += 4;
          }
          else if (strncmp(currentoption, "rgb", 3) ≡ 0) {
            colormodel = PNG_COLOR_TYPE_RGB;
            currentoption += 3;
          }
          else if (strncmp(currentoption, "graya", 5) ≡ 0) {
            colormodel = PNG_COLOR_TYPE_GRAY_ALPHA;
            currentoption += 5;
          }
          else if (strncmp(currentoption, "gray", 4) ≡ 0) {
            colormodel = PNG_COLOR_TYPE_GRAY;
            currentoption += 4;
          }
        }
        else if (strncmp(currentoption, "antialias=", 10) ≡ 0) {
          currentoption += 10;
          if (strncmp(currentoption, "none", 4) ≡ 0) {
            antialias = CAIRO_ANTIALIAS_NONE;
            currentoption += 4;
          }
          else if (strncmp(currentoption, "fast", 4) ≡ 0) {
            antialias = CAIRO_ANTIALIAS_FAST;
            currentoption += 4;
          }
          else if (strncmp(currentoption, "good", 4) ≡ 0) {
            antialias = CAIRO_ANTIALIAS_GOOD;
            currentoption += 4;
          }
          else if (strncmp(currentoption, "best", 4) ≡ 0) {
            antialias = CAIRO_ANTIALIAS_BEST;
            currentoption += 4;
          }
        }
      currentoption = strchr(currentoption, '␣');
      if (currentoption) {
        while (∗currentoption ≡ '␣') currentoption++;
      }
    }
    c = round_unscaled(internal_value(mp_char_code));
    if (standalone) {
      mp→jump_buf = malloc(sizeof(jmp_buf));
      if (mp→jump_buf ≡ Λ ∨ setjmp(∗(mp→jump_buf))) return 0;
    }
    if (mp→history ≥ mp_fatal_error_stop) return 1;
    mp_png_start(mp, hh, number_to_double(internal_value(mp_hppp)),
        number_to_double(internal_value(mp_vppp)), colormodel, antialias);
    p = hh→body; while (p ≠ Λ) {
    if (gr_has_color(p)) mp_png_color_out(mp, p);
    switch (gr_type(p)) { case mp_fill_code: { mp_fill_object ∗ ph = ( mp_fill_object ∗ ) p;
    if (gr_pen_p(ph) ≡ Λ) {
      mp_png_fill_out(mp, gr_path_p(ph), p);
    }
    else if (pen_is_elliptical(gr_pen_p(ph))) {
      pen = mp_png_pen_info(mp, gr_path_p(ph), gr_pen_p(ph));
      mp_png_stroke_out(mp, p, pen, true);
      mp_xfree(pen);
    }
    else {
      mp_png_fill_out(mp, gr_path_p(ph), p);
      mp_png_fill_out(mp, gr_htap_p(ph), p);
    }
    } break; case mp_stroked_code: { mp_stroked_object ∗ ph = ( mp_stroked_object ∗ ) p;
    if (pen_is_elliptical(gr_pen_p(ph))) {
      pen = mp_png_pen_info(mp, gr_path_p(ph), gr_pen_p(ph));
      mp_png_stroke_out(mp, p, pen, false);
      mp_xfree(pen);
    }
    else {
      mp_png_fill_out(mp, gr_path_p(ph), p);
    }
    } break; case mp_text_code: if ((gr_font_n(p) ≠ null_font) ∧ (gr_text_l(p) > 0)) { mp_png_text_out
        (mp, ( mp_text_object ∗ ) p ) ; } break;
  case mp_start_clip_code:
    cairo_save(mp→png→cr); mp_png_path_out (mp, gr_path_p ( ( mp_clip_object ∗ ) p ) ) ;
    cairo_clip(mp→png→cr);
    cairo_new_path(mp→png→cr);
    break;
  case mp_stop_clip_code: cairo_restore(mp→png→cr);
    break;
  case mp_start_bounds_code: case mp_stop_bounds_code: break;
  case mp_special_code: break; }      /∗ all cases are enumerated ∗/
    p = gr_link(p); } (void) mp_set_output_file_name(mp, c);
    mp_store_true_output_filename(mp, c);
    ss = xstrdup(mp→name_of_file);
    cairo_surface_flush(mp→png→surface);
    cairo_destroy(mp→png→cr);
    bitmap.data = cairo_image_surface_get_data(mp→png→surface);
    bitmap.width = cairo_image_surface_get_width(mp→png→surface);
    bitmap.height = cairo_image_surface_get_height(mp→png→surface);
    mp_png_save_to_file(mp, &bitmap, ss, colormodel);
    cairo_surface_destroy(mp→png→surface);
    free(ss);
    return 1; }
```

37. ⟨mplibpng.h 37⟩ ≡

```
#ifndef MPLIBPNG_H
#define MPLIBPNG_H 1
  int mp_png_ship_out(mp_edge_object *hh, const char *options);
#endif
```

38.

```
int mp_png_ship_out(mp_edge_object *hh, const char *options)
  {
    return mp_png_gr_ship_out(hh, options, (int) true);
  }
```