MPlib

API documentation, version 1.800

Taco Hoekwater, September 2012

1 Table of contents

2 Introduction

This document describes the API to MPlib, allowing you to use MPlib in your own applications. One such application is writing bindings to interface with other programming languages. The bindings to the Lua scripting language is part of the MPlib distribution and also covered by this manual.

This is a first draft of both this document as well as the API, so there may be errors or omissions in this document or strangenesses in the API. If you believe something can be improved, please do not hesitate to let us know. The contact email address is `metapost@tug.org`.

The C paragraphs in this document assume you understand C code, the Lua paragraphs assume you understand Lua code, and familiarity with MetaPost is assumed throughout.

2.1 Simple MPlib use

There are two different approaches possible when running MPlib. The first method is most suitable for programs that function as a command-line frontend. It uses 'normal' MetaPost interface with I/O to and from files, and needs very little setup to run. On the other hand, it also gives almost no chance to control the MPlib behaviour.

Here is a C language example of how to do this:

```
#include "mplib.h"
int main (int argc, char **argv) {
  MP mp;
  MP_options *opt = mp_options();
  opt->command_line = argv[1];
  mp = mp_initialize(opt);
  if (mp) {
    int history = mp_run(mp);
    mp_finish(mp);
    exit (history);
  } else {
    exit (EXIT_FAILURE);
  }
}
```

This example will run in 'inimpost' mode. See below for how to preload a macro package.

2.2 Embedded MPlib use

The second method does not run a file, but instead repeatedly executes chunks of MetaPost language input that are passed to the library as strings, with the output redirected to internal buffers instead of directly to files.

Here is an example of how this second approach works, now using the Lua bindings:

```
local mplib = require('mplib')
local mp = mplib.new ({ ini_version = false,
                        mem_name    = 'plain' })
if mp then
```

```
  local l = mp:execute([[beginfig(1);
                         fill fullcircle scaled 20;
                         endfig;
                       ]])
  if l and l.fig and l.fig[1] then
    print (l.fig[1]:postscript())
  end
  mp:finish();
end
```

This example preloads the 'plain' macro file.

3 C API for core MPlib

All of the types, structures, enumerations and functions that are described in this section are defined in the header file `mplib.h`.

3.1 Structures

3.1.1 `MP_options`

This is a structure that contains the configurable parameters for a new MPlib instance. Because MetaPost differentiates between `-ini` and `non-ini` modes, there are three types of settings: Those that apply in both cases, and those that apply in only one of those cases.

int	ini_version	1	set this to zero if you want to load a mem file.
int	error_line	79	maximal length of error message lines
int	half_error_line	50	halfway break point for error contexts
int	max_print_line	100	maximal length of file output
void *	userdata	NULL	for your personal use only, not used by the library
char *	banner	NULL	string to use instead of default banner
int	print_found_names	0	controls whether the asked name or the actual found name of the file is used in messages
int	file_line_error_style	0	when this option is nonzero, the library will use `file:line:error` style formatting for error messages that occur while reading from input files
char *	command_line	NULL	input file name and rest of command line; only used by `mp_run` interface
int	interaction	0	explicit `mp_interaction_mode` (see below)
int	noninteractive	0	set this nonzero to suppress user interaction, only sensible if you want to use `mp_execute`
int	random_seed	0	set this nonzero to force a specific random seed

int	troff_mode	0	set this nonzero to initialize 'troffmode'
char *	mem_name	NULL	explicit mem name to use instead of `plain.mem`. ignored in `-ini` mode.
char *	job_name	NULL	explicit job name to use instead of first input file
mp_file_finder	find_file	NULL	function called for finding files
mp_editor_cmd	run_editor	NULL	function called after 'E' error response
mp_makempx_cmd	run_make_mpx	NULL	function called for the creation of mpx files
int	math_mode	0	set this to `mp_math_double_mode` to use doubles instead of scaled (`mp_math_scaled_mode`) values

To create an `MP_options` structure, you have to use the `mp_options()` function.

3.1.2 `MP`

This type is an opaque pointer to a MPlib instance, it is what you have pass along as the first argument to (almost) all the MPlib functions. The actual C structure it points to has hundreds of fields, but you should not use any of those directly. All configuration is done via the `MP_options` structure, and there are accessor functions for the fields that can be read out.

3.1.3 `mp_run_data`

When the MPlib instance is not interactive, any output is redirected to this structure. There are a few string output streams, and a linked list of output images.

mp_stream	term_out	holds the terminal output
mp_stream	error_out	holds error messages
mp_stream	log_out	holds the log output
mp_stream	ship_out	holds the exported EPS, SVG or PNG string
mp_edge_object *	edges	linked list of generated pictures

`term_out` is equivalent to `stdout` in interactive use, and `error_out` is equivalent to `stderr`. The `error_out` is currently only used for memory allocation errors, the MetaPost error messages are written to `term_out` (and are often duplicated to `log_out` as well).
You need to include at least `mplibps.h` to be able to actually make use of this list of images, see the next section for the details on `mp_edge_object` lists.
See next paragraph for `mp_stream`.

3.1.4 `mp_stream`

This contains the data for a stream as well as some internal bookkeeping variables. The fields that are of interest to you are:

size_t	size	the internal buffer size
char *	data	the actual data.

There is nothing in the stream unless the `size` field is nonzero. There will not be embedded null characters (`\0`) in `data` except when `ship_out` is used for PNG output.
If `size` is nonzero, `strlen(data)` is guaranteed to be less than that, and may be as low as zero (if MPlib has written an empty string).

3.2 Function prototype typedefs

The following three function prototypes define functions that you can pass to MPlib inside the `MP_options` structure.

3.2.1 `char * (*mp_file_finder) (MP, const char*, const char*, int)`

MPlib calls this function whenever it needs to find a file. If you do not set up the matching option field (`MP_options.find_file`), MPlib will only be able to find files in the current directory.
The three function arguments are the requested file name, the file mode (either `"r"` or `"w"`), and the file type (an `mp_filetype`, see below).
The return value is a new string indicating the disk file name to be used, or NULL if the named file can not be found. If the mode is `"w"`, it is usually best to simply return a copy of the first argument.

3.2.2 `void (*mp_editor_cmd)(MP, char*, int)`

This function is executed when a user has pressed 'E' as reply to an MetaPost error, so it will only ever be called when MPlib in interactive mode. The function arguments are the file name and the line number. When this function is called, any open files are already closed.

3.2.3 `int (*mp_makempx_cmd)(MP, char*, char *)`

This function is executed when there is a need to start generating an `mpx` file because (the first time a `btex` command was encountered in the current input file).
The first argument is the input file name. This is the name that was given in the MetaPost language, so it may not be the same as the name of the actual file that is being used, depending on how your `mp_file_finder` function behaves. The second argument is the requested output name for mpx commands.
A zero return value indicates success, everything else indicates failure to create a proper `mpx` file and will result in an MetaPost error.

3.3 Enumerations

3.3.1 `mp_filetype`

The `mp_file_finder` receives an `int` argument that is one of the following types:

mp_filetype_program	Metapost language code (r)
mp_filetype_log	Log output (w)

mp_filetype_postscript	PostScript or SVG output (w)
mp_filetype_bitmap	PNG output (w)
mp_filetype_metrics	TEX font metric file (r+w)
mp_filetype_fontmap	Font map file (r)
mp_filetype_font	Font PFB file (r)
mp_filetype_encoding	Font encoding file (r)
mp_filetype_text	`readfrom` and `write` files (r+w)

3.3.2 `mp_interaction_mode`

When `noninteractive` is zero, MPlib normally starts in a mode where it reports every error, stops and asks the user for input. This initial mode can be overruled by using one of the following:

mp_batch_mode	as with `batchmode`
mp_nonstop_mode	as with `nonstopmode`
mp_scroll_mode	as with `scrollmode`
mp_error_stop_mode	as with `errorstopmode`

3.3.3 `mp_math_mode`

mp_math_scaled_mode	uses scaled point data for numerical values
mp_math_double_mode	uses IEEE double floating point data for numerical values
mp_math_binary_mode	not used yet.
mp_math_decimal_mode	not used yet.

3.3.4 `mp_history_state`

These are set depending on the current state of the interpreter.

mp_spotless	still clean as a whistle
mp_warning_issued	a warning was issued or something was `show`-ed
mp_error_message_issued	an error has been reported
mp_fatal_eror_stop	termination was premature due to error(s)
mp_system_error_stop	termination was premature due to disaster (out of system memory)

3.3.5 `mp_color_model`

Graphical objects always have a color model attached to them.

mp_no_model	as with `withoutcolor`
mp_grey_model	as with `withgreycolor`
mp_rgb_model	as with `withrgbcolor`
mp_cmyk_model	as with `withcmykcolor`

3.3.6 `mp_graphical_object_code`

There are eight different graphical object types.

mp_fill_code	`addto contour`
mp_stroked_code	`addto doublepath`
mp_text_code	`addto also` (via `infont`)
mp_start_clip_code	`clip`
mp_start_bounds_code	
mp_stop_clip_code	`setbounds`
mp_stop_bounds_code	
mp_special_code	`special`

3.4 Functions

3.4.1 `char * mp_metapost_version(void)`

Returns a copy of the MPlib version string.

3.4.2 `MP_options * mp_options(void)`

Returns a properly initialized option structure, or `NULL` in case of allocation errors.

3.4.3 `MP mp_initialize(MP_options *opt)`

Returns a pointer to a new MPlib instance, or `NULL` if initialisation failed.
String options are copied, so you can free any of those (and the `opt` structure) immediately after the call to this function.

3.4.4 `int mp_status(MP mp)`

Returns the current value of the interpreter error state, as a `mp_history_state`. This function is useful after `mp_initialize`.

3.4.5 `int mp_run(MP mp)`

Runs the MPlib instance using the `command_line` and other items from the `MP_options`. After the call to `mp_run`, the MPlib instance should be closed off by calling `mp_finish`.
The return value is the current `mp_history_state`

3.4.6 `void * mp_userdata(MP mp)`

Simply returns the pointer that was passed along as `userdata` in the `MP_options` struct.

3.4.7 `int mp_troff_mode(MP mp)`

Returns the value of `troff_mode` as copied from the `MP_options` struct.

3.4.8 `mp_run_data * mp_rundata(MP mp)`

Returns the information collected during the previous call to `mp_execute`.

3.4.9 `int mp_execute(MP mp, char *s, size_t l)`

Executes string `s` with length `l` in the MPlib instance. This call can be repeated as often as is needed. The return value is the current `mp_history_state`. To get at the produced results, call `mp_rundata`.

3.4.10 `void mp_finish(MP mp)`

This finishes off the use of the MPlib instance: it closes all files and frees all the memory allocated by this instance.

3.4.11 `double mp_get_char_dimension(MP mp,char*fname,int n,int t)`

This is a helper function that returns one of the dimensions of glyph `n` in font `fname` as a double in PostScript (AFM) units. The requested item `t` can be `'w'` (width), `'h'` (height), or `'d'` (depth).

3.4.12 `int mp_memory_usage(MP mp)`

Returns the current memory usage of this instance.

3.4.13 `int mp_hash_usage(MP mp)`

Returns the current hash usage of this instance.

3.4.14 `int mp_param_usage(MP mp)`

Returns the current simultaneous macro parameter usage of this instance.

3.4.15 `int mp_open_usage(MP mp)`

Returns the current `input` levels of this instance.

4 C API for path and knot manipulation

4.1 Enumerations

4.1.1 `mp_knot_type`

Knots can have left and right types depending on their current status. By the time you see them in the output, they are usually either `mp_explicit` or `mp_endpoint`, but here is the full list:

mp_endpoint
mp_explicit
mp_given
mp_curl
mp_open
mp_end_cycle

4.1.2 `mp_knot_originator`

Knots can originate from two sources: they can be explicitly given by the user, or they can be created by the MPlib program code (for example as result of the `makepath` operator).

mp_program_code
mp_metapost_user

4.2 Structures

4.2.1 `mp_number`

Numerical values are represented by opaque structure pointers named `mp_number`.

4.2.2 `mp_knot`

Each MPlib path (a sequence of MetaPost points) is represented as a linked list of structure pointers of the type `mp_knot`.

mp_knot	next	the next knot, or NULL
mp_knot_type	data.types.left_type	the `mp_knot_type` for the left side
mp_knot_type	data.types.right_type	the `mp_knot_type` for the right side
mp_number	x_coord	x
mp_number	y_coord	y
mp_number	left_x	x of the left (incoming) control point
mp_number	left_y	y of the left (incoming) control point
mp_number	right_x	x of the right (outgoing) control point
mp_number	right_y	y of the right (outgoing) control point
mp_knot_originator	originator	the `mp_knot_originator`

Paths are always represented as a circular list. The difference between cyclic and non-cyclic paths is indicated by their `mp_knot_type`.
While the fields of the knot structure are in fact accessible, it is better to use the access functions below as the internal structure tends to change.

4.3 Functions for accessing knot data

4.3.1 `mp_number mp_knot_x_coord(MP mp,mp_knot p)`

Access the x coordinate of the knot.

4.3.2 `mp_number mp_knot_y_coord(MP mp,mp_knot p)`

Access the y coordinate of the knot.

4.3.3 `mp_number mp_knot_left_x(MP mp,mp_knot p)`

Access the x coordinate of the left control point of the knot.

4.3.4 `mp_number mp_knot_left_y(MP mp,mp_knot p)`

Access the y coordinate of the left control point of the knot.

4.3.5 `mp_number mp_knot_right_x(MP mp,mp_knot p)`

Access the x coordinate of the right control point of the knot.

4.3.6 `mp_number mp_knot_right_y(MP mp,mp_knot p)`

Access the y coordinate of the right control point of the knot.

4.3.7 `int mp_knot_left_type(MP mp,mp_knot p)`

Access the type of the knot on the left side.

4.3.8 `int mp_knot_right_type(MP mp,mp_knot p)`

Access the type of the knot on the right side.

4.3.9 `mp_knot mp_knot_next(MP mp,mp_knot p)`

Access the pointer to the next knot.

4.3.10 `mp_number mp_knot_left_curl(MP mp,mp_knot p)`

Access the left curl of the knot (applies to unresolved knots, see below).

4.3.11 `mp_number mp_knot_left_given(MP mp,mp_knot p)`

Access the left given value of the knot (applies to unresolved knots, see below).

4.3.12 `mp_number mp_knot_left_tension(MP mp,mp_knot p)`

Access the left tension of the knot (applies to unresolved knots, see below).

4.3.13 `mp_number mp_knot_right_curl(MP mp,mp_knot p)`

Access the right curl value of the knot (applies to unresolved knots, see below).

4.3.14 `mp_number mp_knot_right_given(MP mp,mp_knot p)`

Access the right given value of the knot (applies to unresolved knots, see below).

4.3.15 `mp_number mp_knot_right_tension(MP mp,mp_knot p)`

Access the right tension value of the knot (applies to unresolved knots, see below).

4.3.16 `double mp_number_as_double(MP mp,mp_number n)`

Converts an `mp_number` to `double`.

4.4 Functions for creating and modifying knot data

4.4.1 `mp_knot mp_create_knot(MP mp)`

Allocates and returns a new knot. Returns `NULL` on (malloc) failure.

4.4.2 `int mp_set_knot(MP mp,mp_knot p,double x,double y)`

Fills in the coordinate of knot p. `x1` and `y1` values should be within the proper range for the current numerical mode. Return 1 on success, 0 on failure.

4.4.3 `int mp_close_path(MP mp,mp_knot p,mp_knot q)`

Connects p and q using an 'endpoint join', where p is the last knot of the path, and q is the first knot. The right tension of p and the left tension of q are (re)set to the default of 1.0.
Because all knot list data structures are always circular, this is needed to end the path properly even if the path is not intended cyclic (or use `mp_close_path_cycle()`, if it is indeed a cycle). Return 1 on success, 0 on failure.

4.4.4 `int mp_close_path_cycle(MP mp,mp_knot p,mp_knot q)`

Connects p and q using an 'open join', where p is the last knot of the path, and q is the first knot. The right tension of p and the left tension of q are (re)set to the default of 1.0.
This is needed to mimic metapost's `cycle`. return 1 on success, 0 on failure.

4.4.5 `mp_knot mp_append_knot(MP mp,mp_knot p,double x,double y)`

Appends a knot to previous knot q, and returns the new knot. This is a convenience method combining `mp_create_knot()`, `mp_set_knot()`, and (if q is not NULL) `mp_close_path_cycle()`. Returns NULL on failure.

4.4.6 `int mp_set_knot_left_curl(MP mp,mp_knot q,double value)`

Sets the left curl value for a knot. `fabs(value)` should be less than 4096.0 return 1 on success, 0 on failure.

4.4.7 `int mp_set_knot_right_curl(MP mp,mp_knot q,double value)`

Sets the right curl value for a knot. `fabs(value)` should be less than 4096.0 return 1 on success, 0 on failure.

4.4.8 `int mp_set_knot_curl(MP mp,mp_knot q,double value)`

Sets the curl value for a knot. `fabs(value)` should be less than 4096.0 return 1 on success, 0 on failure.

4.4.9 `int mp_set_knotpair_curls(MP mp,mp_knot p,mp_knot q,double t1,double t2)`

A convenience method that calls `mp_set_knot_curl(mp,p,t1)` and `mp_set_knot_curl(mp,q,t2)` return 1 if both succeed, 0 otherwise.

4.4.10 `int mp_set_knot_direction(MP mp,mp_knot q,double x,double y)`

Sets the direction `{x,y}` value for a knot. `fabs(x)` and `fabs(y)` should be less than 4096.0 return 1 on success, 0 on failure.

4.4.11 `int mp_set_knotpair_directions(MP mp,mp_knot p,mp_knot q,double x1,double y1,double x2,double y2)`

A convenience method that calls `mp_set_knot_direction(mp,p,x1,y1)` and `mp_set_knot_direction(mp,p,x2,y2`
return 1 if both succeed, 0 otherwise.

4.4.12 `int mp_set_knotpair_tensions(MP mp,mp_knot p,mp_knot q,double t1,double t2)`

Sets the tension specifiers for a pair of connected knots. `fabs(t1)` and `fabs(t2)` should be more than 0.75 and less than 4096.0 return 1 on success, 0 on failure.

4.4.13 `int mp_set_knot_left_tension(MP mp, mp_knot p, double t1)`

Set the left tension of a knot. `fabs(t1)` should be more than 0.75 and less than 4096.0 return 1 on success, 0 on failure.

4.4.14 `int mp_set_knot_right_tension(MP mp, mp_knot p, double t1)`

Set the right tension of a knot. `fabs(t1)` should be more than 0.75 and less than 4096.0 return 1 on success, 0 on failure.

4.4.15 `int mp_set_knot_left_control(MP mp, mp_knot p, double x1, double y1)`

4.4.16 `int mp_set_knot_right_control(MP mp, mp_knot p, double x1, double y1)`

Sets explicit left or right control for a knot. `x1` and `y1` values should be within the proper range for the current numerical mode. return 1 on success, 0 on failure.

4.4.17 `int mp_set_knotpair_controls(MP mp,mp_knot p,mp_knot q,double x1,double y1,double x2,double y2)`

Sets explicit controls for a knot pair. All four `x` and `y` values should be within the proper range for the current numerical mode. return 1 on success, 0 on failure.

4.4.18 `int mp_solve_path(MP mp,mp_knot first)`

Finds explicit controls for the knot list at `first`, which is changed in-situ. Returns 0 if there was any kind of error, in which case `first` is unmodified. There can be quite a set of potential errors, mostly harmless processing errors. However, be aware that it is also possible that there are internal mplib memory allocation errors. A distinction between those can not be made at the moment. Return 1 on success, 0 on failure.

4.4.19 `void mp_free_path(MP mp,mp_knot p)`

Frees the memory of a path.

4.5 Example usage

Since the above function list is quite dry and not that easy to grasp, here are two examples of how to use it. First a simple example (`mp_dump_path()` code is given below).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mplib.h"

int main (int argc, char ** argv)  {
        MP mp ;
        mp_knot p, first, q;
        MP_options * opt = mp_options () ;
        opt -> command_line = NULL;
        opt -> noninteractive = 1 ;
        mp = mp_initialize ( opt ) ;
        if ( ! mp ) exit ( EXIT_FAILURE ) ;
        /* Equivalent Metapost code:

           path p;
           p := (0,0)..(10,10)..(10,-5)..cycle;

         */
        first = p = mp_append_knot(mp,NULL,0,0);
        if ( ! p ) exit ( EXIT_FAILURE ) ;
        q = mp_append_knot(mp,p,10,10);
        if ( ! q ) exit ( EXIT_FAILURE ) ;
        p = mp_append_knot(mp,q,10,-5);
        if ( ! p ) exit ( EXIT_FAILURE ) ;
        mp_close_path_cycle(mp, p, first);
        /* mp_dump_path(mp, first); */
        if (mp_solve_path(mp, first)) {
            /* mp_dump_path(mp, first); */
        }
        mp_free_path(mp, first);
        mp_finish ( mp ) ;
        free(opt);
        return 0;
}
```

For some more challenging path input, here is a more elaborate example of the path processing code:

```
        /* Equivalent Metapost code:

           path p;
           p := (0,0)..
                (2,20)--
                (10, 5)..controls (2,2) and (9,4.5)..
                (3,10)..tension 3 and atleast 4 ..
                (1,14){2,0} .. {0,1}(5,-4);
         */
        first = p = mp_append_knot(mp,NULL,0,0);
        q = mp_append_knot(mp,p,2,20);
        p = mp_append_knot(mp,q,10,5);
        if (!mp_set_knotpair_curls(mp, q,p, 1.0, 1.0))
          exit ( EXIT_FAILURE ) ;
        q = mp_append_knot(mp,p,3,10);
        if (!mp_set_knotpair_controls(mp, p,q, 2.0, 2.0, 9.0, 4.5))
          exit ( EXIT_FAILURE ) ;
        p = mp_append_knot(mp,q,1,14);
        if (!mp_set_knotpair_tensions(mp,q,p, 3.0, -4.0))
          exit ( EXIT_FAILURE ) ;
        q = mp_append_knot(mp,p,5,-4);
        if (!mp_set_knotpair_directions(mp, p,q, 2.0, 0.0, 0.0, 1.0))
```

```
          exit ( EXIT_FAILURE ) ;

        mp_close_path(mp, q, first);

        /* mp_dump_path(mp, first); */
        if (mp_solve_path(mp, first)) {
            /* mp_dump_path(mp, first); */
        }
        mp_free_path(mp, first);
```

And here is the source code for the `mp_dump_path` function, which produces path output that is similar to Metapost's `tracingchoices` report.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mplib.h"

#define ROUNDED_ZERO(v) (fabs((v))<0.00001 ? 0 : (v))
#define PI 3.1415926535897932384626433832795028841971
#define RADIANS(a) (mp_number_as_double(mp,(a)) / 16.0) * PI/180.0

void mp_dump_path (MP mp, mp_knot h) {
  mp_knot p, q;
  if (h == NULL) return;
  p = h;
  do {
    q=mp_knot_next(mp,p);
    if ( (p==NULL)||(q==NULL) ) {
      printf("\n???");
      return; /* this won't happen */
    }
    printf ("(%g,%g)", mp_number_as_double(mp,mp_knot_x_coord(mp,p)),
                       mp_number_as_double(mp,mp_knot_y_coord(mp,p)));
    switch (mp_knot_right_type(mp,p)) {
    case mp_endpoint:
      if ( mp_knot_left_type(mp,p)==mp_open ) printf("{open?}");
      if ( (mp_knot_left_type(mp,q)!=mp_endpoint)||(q!=h) )
        q=NULL; /* force an error */
      goto DONE;
      break;
    case mp_explicit:
      printf ("..controls (%g,%g)",
              mp_number_as_double(mp,mp_knot_right_x(mp,p)),
              mp_number_as_double(mp,mp_knot_right_y(mp,p)));
      printf(" and ");
      if ( mp_knot_left_type(mp,q)!=mp_explicit ) {
        printf("??");
      } else {
        printf ("(%g,%g)",mp_number_as_double(mp,mp_knot_left_x(mp,q)),
                         mp_number_as_double(mp,mp_knot_left_y(mp,q)));
      }
      goto DONE;
      break;
    case mp_open:
      if ( (mp_knot_left_type(mp,p)!=mp_explicit)
           &&
           (mp_knot_left_type(mp,p)!=mp_open) ) {
        printf("{open?}");
      }
      break;
    case mp_curl:
    case mp_given:
      if ( mp_knot_left_type(mp,p)==mp_open )
        printf("??");
      if ( mp_knot_right_type(mp,p)==mp_curl ) {
        printf("{curl %g}", mp_number_as_double(mp,mp_knot_right_curl(mp,p)));
      } else {
        double rad = RADIANS(mp_knot_right_curl(mp,p));
        double n_cos = ROUNDED_ZERO(cos(rad)*4096);
        double n_sin = ROUNDED_ZERO(sin(rad)*4096);
        printf("{%g,%g}", n_cos, n_sin);
      }
      break;
    }
    if ( mp_knot_left_type(mp,q)<=mp_explicit ) {
      printf("..control?"); /* can't happen */
    } else if ((mp_number_as_double(mp,mp_knot_right_tension(mp,p))!=(1.0))||
               (mp_number_as_double(mp,mp_knot_left_tension(mp,q)) !=(1.0)))
{
      printf("..tension ");
      if ( mp_number_as_double(mp,mp_knot_right_tension(mp,p))<0.0 )
        printf("atleast ");
      printf("%g", fabs(mp_number_as_double(mp,mp_knot_right_tension(mp,p))));
      if (mp_number_as_double(mp,mp_knot_right_tension(mp,p)) !=
          mp_number_as_double(mp,mp_knot_left_tension(mp,q))) {
        printf(" and ");
        if (mp_number_as_double(mp,mp_knot_left_tension(mp,q))< 0.0)
          printf("atleast ");
        printf("%g", fabs(mp_number_as_double(mp,mp_knot_left_tension(mp,q))));
      }
    }
   DONE:
    p=q;
    if ( p!=h || mp_knot_left_type(mp,h)!=mp_endpoint) {
      printf ("\n ..");
      if ( mp_knot_left_type(mp,p) == mp_given ) {
        double rad = RADIANS(mp_knot_left_curl(mp,p));
        double n_cos = ROUNDED_ZERO(cos(rad)*4096);
        double n_sin = ROUNDED_ZERO(sin(rad)*4096);
        printf("{%g,%g}", n_cos, n_sin);
      } else if ( mp_knot_left_type(mp,p) ==mp_curl ){
        printf("{curl %g}", mp_number_as_double(mp,mp_knot_left_curl(mp,p)));
      }
    }
  } while (p!=h);
  if ( mp_knot_left_type(mp,h)!=mp_endpoint )
    printf("cycle");
  printf (";\n");
}
```

The above function is much complicated because of all the knot type cases that can only happen *before* `mp_solve_path()` is called. A version that only prints processed paths and is less scared of using direct field access would be much shorter:

```
void mp_dump_solved_path (MP mp, mp_knot h) {
    mp_knot p, q;
    if (h == NULL) return;
    p = h;
    do {
        q=mp_knot_next(mp,p);
        printf ("(%g,%g)..controls (%g,%g) and (%g,%g)",
                mp_number_as_double(mp,p->x_coord),
                mp_number_as_double(mp,p->y_coord),
                mp_number_as_double(mp,p->right_x),
                mp_number_as_double(mp,p->right_y),
                mp_number_as_double(mp,q->left_x),
                mp_number_as_double(mp,q->left_y));
        p=q;
        if ( p!=h || h->data.types.left_type!=mp_endpoint) {
            printf ("\n ..");
        }
    } while (p!=h);
    if (  h->data.types.left_type!=mp_endpoint )
        printf("cycle");
    printf (";\n");
}
```

5 C API for graphical backend functions

These are all defined in `mplibps.h`

5.1 Structures

The structures in this section are used by the items in the body of the `edges` field of an `mp_rundata` structure. They are presented here in a bottom-up manner.

5.1.1 `mp_gr_knot`

These are like `mp_knot`, except that all `mp_number` values have been simplified to `double`.

5.1.2 `mp_color`

The graphical object that can be colored, have two fields to define the color: one for the color model and one for the color values. The structure for the color values is defined as follows:

double	a_val	see below
double	b_val	–
double	c_val	–
double	d_val	–

All graphical objects that have `mp_color` fields also have `mp_color_model` fields. The color model decides the meaning of the four data fields:

color model value	a_val	b_val	c_val	d_val
mp_no_model	–	–	–	–
mp_grey_model	grey	–	–	–
mp_rgb_model	red	green	blue	
mp_cmyk_model	cyan	magenta	yellow	black

5.1.3 `mp_dash_object`

Dash lists are represented like this:

double *	array	an array of dash lengths, terminated by -1.
double	offset	the dash array offset (as in PostScript)

5.1.4 `mp_graphic_object`

Now follow the structure definitions of the objects that can appear inside a figure (this is called an 'edge structure' in the internal WEB documentation).
There are eight different graphical object types, but there are seven different C structures. Type `mp_graphic_object` represents the base line of graphical object types. It has only two fields:

mp_graphical_object_code	type	
struct mp_graphic_object *	next	next object or NULL

Because every graphical object has at least these two fields, the body of a picture is represented as a linked list of `mp_graphic_object` items. Each object in turn can then be typecast to the proper type depending on its `type`.
The two 'missing' objects in the explanations below are the ones that match `mp_stop_clip_code` and `mp_stop_bounds_code`: these have no extra fields besides `type` and `next`.

5.1.5 `mp_fill_object`

Contains the following fields on top of the ones defined by `mp_graphic_object`:

char *	pre_script	this is the result of `withprescript`
char *	post_script	this is the result of `withpostscript`
mp_color	color	the color value of this object
mp_color_model	color_model	the color model
unsigned char	ljoin	the line join style; values have the same meaning as in PostScript: 0 for mitered, 1 for round, 2 for beveled.
mp_gr_knot	path_p	the (always cyclic) path
mp_gr_knot	htap_p	a possible reversed path (see below)
mp_gr_knot	pen_p	a possible pen (see below)
double	miterlim	the miter limit

Even though this object is called an `mp_fill_object`, it can be the result of both `fill` and `filldraw` in the MetaPost input. This means that there can be a pen involved as well. The final output should behave as follows:

- If there is no `pen_p`; simply fill `path_p`.
- If there is a one-knot pen (`pen_p->next` = `pen_p`) then fill `path_p` and also draw `path_p` with the `pen_p`. Do not forget to take `ljoin` and `miterlim` into account when drawing with the pen.
- If there is a more complex pen (`pen_p->next` != `pen_p`) then its path has already been preprocessed for you: `path_p` and `htap_p` already incorporate its shape.

5.1.6 `mp_stroked_object`

Contains the following fields on top of the ones defined by `mp_graphic_object`:

char *	pre_script	this is the result of `withprescript`
char *	post_script	this is the result of `withpostscript`
mp_color	color	color value
mp_color_model	color_model	color model
unsigned char	ljoin	the line join style
unsigned char	lcap	the line cap style; values have the same meaning as in PostScript: 0 for butt ends, 1 for round ends, 2 for projecting ends.

mp_gr_knot	path_p	the path
mp_gr_knot	pen_p	the pen
double	miterlim	miter limit
mp_dash_object *	dash_p	a possible dash list

5.1.7 `mp_text_object`

Contains the following fields on top of the ones defined by `mp_graphic_object`:

char *	pre_script	this is the result of `withprescript`
char *	post_script	this is the result of `withpostscript`
mp_color	color	color value
mp_color_model	color_model	color model
char *	text_p	string to be placed
char *	font_name	the MetaPost font name
double	font_dsize	size of the font
double	width	width of the picture resulting from the string
double	height	height
double	depth	depth
double	tx	transformation component
double	ty	transformation component
double	txx	transformation component
double	tyx	transformation component
double	txy	transformation component
double	tyy	transformation component

All fonts are loaded by MPlib at the design size (but not all fonts have the same design size). If text is to be scaled, this happens via the transformation components.

5.1.8 `mp_clip_object`

Contains the following field on top of the ones defined by `mp_graphic_object`:

mp_gr_knot	path_p	defines the clipping path that is in effect until the object with the matching `mp_stop_clip_code` is encountered

5.1.9 `mp_bounds_object`

Contains the following field on top of the ones defined by `mp_graphic_object`:

mp_gr_knot	path_p	the path that was used for boundary calculation

This object can be ignored when output is generated, it only has effect on the boudingbox of the following objects and that has been taken into account already.

5.1.10 `mp_special_object`

This represents the output generated by a MetaPost `special` command. It contains the following field on top of the ones defined by `mp_graphic_object`:

char *	pre_script	the special string

Each `special` command generates one object. All of the relevant `mp_special_objects` for a figure are linked together at the start of that figure.

5.1.11 `mp_edge_object`

mp_edge_object *	next	points to the next figure (or NULL)
mp_graphic_object *	body	a linked list of objects in this figure
char *	filename	this would have been the used filename if a PostScript file would have been generated
MP	parent	a pointer to the instance that created this figure
double	minx	lower-left x of the bounding box
double	miny	lower-left y of the bounding box
double	maxx	upper right x of the bounding box
double	maxy	upper right y of the bounding box
double	width	value of `charwd`; this would become the TFM width (but without the potential rounding correction for TFM file format)
double	height	similar for height (`charht`)
double	depth	similar for depth (`chardp`)
double	ital_corr	similar for italic correction (`charic`)
int	charcode	Value of `charcode` (rounded, but not modulated for TFM's 256 values yet)

5.2 Functions

5.2.1 `int mp_ps_ship_out(mp_edge_object*hh,int prologues,int procset)`

If you have an `mp_edge_object`, you can call this function. It will generate the PostScript output for the figure and save it internally. A subsequent call to `mp_rundata` will find the generated text in the `ship_out` field.
Returns zero for success.

5.2.2 `int mp_svg_ship_out(mp_edge_object*hh,int prologues)`

If you have an `mp_edge_object`, you can call this function. It will generate the SVG output for the figure and save it internally. A subsequent call to `mp_rundata` will find the generated text in the `ship_out` field.
Returns zero for success.

5.2.3 `int mp_png_ship_out(mp_edge_object*hh,int prologues)`

If you have an `mp_edge_object`, you can call this function. It will generate the PNG bitmap for the figure and save it internally. A subsequent call to `mp_rundata` will find the generated data in the `ship_out` field.
Note: the `prologues` argument is currently unused, but it may be used in the future to support MNG output.
Returns zero for success.

5.2.4 `void mp_gr_toss_objects(mp_edge_object*hh)`

This frees a single `mp_edge_object` and its `mp_graphic_object` contents.

5.2.5 `void mp_gr_toss_object(mp_graphic_object*p)`

This frees a single `mp_graphic_object` object.

5.2.6 `mp_graphic_object * mp_gr_copy_object(MP mp,mp_graphic_object*p)`

This creates a deep copy of a `mp_graphic_object` object.

6 C API for label generation (a.k.a. makempx)

The following are all defined in `mpxout.h`.

6.1 Structures

6.1.1 `MPX`

An opaque pointer that is passed on to the file_finder.

6.1.2 `mpx_options`

This structure holds the option fields for `mpx` generation. You have to fill in all fields except `mptexpre`, that one defaults to `mptexpre.tex`

mpx_modes	mode	
char *	cmd	the command (or sequence of commands) to run
char *	mptexpre	prepended to the generated TEX file
char *	mpname	input file name
char *	mpxname	output file name
char *	banner	string to be printed to the generated to-be-typeset file
int	debug	When nonzero, `mp_makempx` outputs some debug information and do not delete temp files
mpx_file_finder	find_file	

6.2 Function prototype typedefs

6.2.1 `char * (*mpx_file_finder) (MPX, const char*, const char*, int)`

The return value is a new string indicating the disk file to be used. The arguments are the file name, the file mode (either `"r"` or `"w"`), and the file type (an `mpx_filetype`, see below). If the mode is `"w"`, it is usually best to simply return a copy of the first argument.

6.3 Enumerations

6.3.1 `mpx_modes`

mpx_tex_mode
mpx_troff_mode

6.3.2 `mpx_filetype`

mpx_tfm_format	TeX or Troff ffont metric file
mpx_vf_format	TeX virtual font file
mpx_trfontmap_format	Troff font map
mpx_trcharadj_format	Troff character shift information
mpx_desc_format	Troff DESC file
mpx_fontdesc_format	Troff FONTDESC file
mpx_specchar_format	Troff special character definition

6.4 Functions

6.4.1 `int mpx_makempx(mpx_options *mpxopt)`

A return value of zero is success, non-zero values indicate errors.

7 Lua API

The MetaPost library interface registers itself in the table `mplib`.

7.1 mplib.version

Returns the MPlib version.

```
<string> s = mplib.version()
```

7.2 mplib.new

To create a new metapost instance, call

```
<mpinstance> mp = mplib.new({...})
```

This creates the `mp` instance object. The `mp` instance object always starts out in so-called 'inimp' mode, there is no support for preload files.
The argument hash can have a number of different fields, as follows:

name	**type**	**description**	**default**
`error_line`	number	error line width	79
`print_line`	number	line length in ps output	100
`random_seed`	number	the initial random seed	variable
`interaction`	string	the interaction mode, one of `batch`, `nonstop`, `scroll`, `errorstop`	`errorstop`
`job_name`	string	`--jobname`	`mpout`
`math_mode`	string	the number system mode, one of `scaled` or `double`	`scaled`
`find_file`	function	a function to find files	only local files

The `find_file` function should be of this form:

```
<string> found = finder (<string> name, <string> mode, <string> type)
```

with:

`name`	the requested file
`mode`	the file mode: `r` or `w`
`type`	the kind of file, one of: `mp`, `tfm`, `map`, `pfb`, `enc`

Return either the full pathname of the found file, or `nil` if the file cannot be found.

7.3 mp:statistics

You can request statistics with:

```
<table> stats = mp:statistics()
```

This function returns the allocation statistics for an MPlib instance. There are four fields, giving the maximum number of used items in each of four object classes:

`memory`	number	allocated memory (in bytes)
`hash`	number	hash size (in entries)
`params`	number	simultaneous macro parameters
`open`	number	input file nesting levels

7.4 mp:execute

You can ask the METAPOST interpreter to run a chunk of code by calling

```
local rettable = mp:execute('metapost language chunk')
```

for various bits of Metapost language input. Be sure to check the `rettable.status` (see below) because when a fatal METAPOST error occurs the MPlib instance will become unusable thereafter. Generally speaking, it is best to keep your chunks small, but beware that all chunks have to obey proper syntax, like each of them is a small file. For instance, you cannot split a single statement over multiple chunks.

In contrast with the normal standalone `mpost` command, there is *no* implied 'input' at the start of the first chunk.

7.5 mp:finish

```
local rettable = mp:finish()
```

If for some reason you want to stop using an MPlib instance while processing is not yet actually done, you can call `mp:finish`. Eventually, used memory will be freed and open files will be closed by the Lua garbage collector, but an explicit `mp:finish` is the only way to capture the final part of the output streams.

7.6 Result table

The return value of `mp:execute` and `mp:finish` is a table with a few possible keys (only `status` is always guaranteed to be present).

log	string	output to the 'log' stream
term	string	output to the 'term' stream
error	string	output to the 'error' stream (only used for 'out of memory')
status	number	the return value: 0=good, 1=warning, 2=errors, 3=fatal error
fig	table	an array of generated figures (if any)

When `status` equals 3, you should stop using this MPlib instance immediately, it is no longer capable of processing input.

If it is present, each of the entries in the `fig` array is a userdata representing a figure object, and each of those has a number of object methods you can call:

boundingbox	function	returns the bounding box, as an array of 4 values
postscript	function	return a string that is the ps output of the `fig`
svg	function	return a string that is the svg output of the `fig`
png	function	return a string that is the png output of the `fig`
objects	function	returns the actual array of graphic objects in this `fig`
copy_objects	function	returns a deep copy of the array of graphic objects in this `fig`
filename	function	the filename this `fig`'s PostScript output would have written to in standalone mode
width	function	the `charwd` value
height	function	the `charht` value
depth	function	the `chardp` value
italcorr	function	the `charic` value
charcode	function	the (rounded) `charcode` value

NOTE: you can call `fig:objects()` only once for any one `fig` object!
When the boundingbox represents a 'negated rectangle', i.e. when the first set of coordinates is larger than the second set, the picture is empty.
Graphical objects come in various types that each have a different list of accessible values. The types are: `fill`, `outline`, `text`, `start_clip`, `stop_clip`, `start_bounds`, `stop_bounds`, `special`.
There is helper function (`mplib.fields(obj)`) to get the list of accessible values for a particular object, but you can just as easily use the tables given below).
All graphical objects have a field `type` that gives the object type as a string value, that not explicit mentioned in the tables. In the following, `numbers` are PostScript points represented as a floating point number, unless stated otherwise. Field values that are of `table` are explained in the next section.

7.6.1 fill

path	table	the list of knots
htap	table	the list of knots for the reversed trajectory
pen	table	knots of the pen
color	table	the object's color
linejoin	number	line join style (bare number)
miterlimit	number	miter limit
prescript	string	the prescript text
postscript	string	the postscript text

The entries `htap` and `pen` are optional.
There is helper function (`mplib.pen_info(obj)`) that returns a table containing a bunch of vital characteristics of the used pen (all values are floats):

width	number	width of the pen
rx	number	x scale
sx	number	xy multiplier
sy	number	yx multiplier
ry	number	y scale
tx	number	x offset
ty	number	y offset

7.6.2 outline

path	table	the list of knots
pen	table	knots of the pen
color	table	the object’s color
linejoin	number	line join style (bare number)
miterlimit	number	miter limit
linecap	number	line cap style (bare number)
dash	table	representation of a dash list
prescript	string	the prescript text
postscript	string	the postscript text

The entry `dash` is optional.

7.6.3 text

text	string	the text
font	string	font tfm name
dsize	number	font size
color	table	the object’s color
width	number	
height	number	
depth	number	
transform	table	a text transformation
prescript	string	the prescript text
postscript	string	the postscript text

7.6.4 special

prescript	string	special text

7.6.5 start_bounds, start_clip

path	table	the list of knots

7.6.6 stop_bounds, stop_clip

Here are no fields available.

7.7 Subsidiary table formats

7.7.1 Paths and pens

Paths and pens (that are really just a special type of paths as far as MPlib is concerned) are represented by an array where each entry is a table that represents a knot.

`left_type`	string	when present: 'endpoint', but ususally absent
`right_type`	string	like `left_type`
`x_coord`	number	x coordinate of this knot
`y_coord`	number	y coordinate of this knot
`left_x`	number	x coordinate of the precontrol point of this knot
`left_y`	number	y coordinate of the precontrol point of this knot
`right_x`	number	x coordinate of the postcontrol point of this knot
`right_y`	number	y coordinate of the postcontrol point of this knot

There is one special case: pens that are (possibly transformed) ellipses have an extra string-valued key `type` with value `elliptical` besides the array part containing the knot list.

7.7.2 Colors

A color is an integer array with 0, 1, 3 or 4 values:

0	marking only	no values
1	greyscale	one value in the range (0,1), 'black' is 0
3	RGB	three values in the range (0,1), 'black' is 0,0,0
4	CMYK	four values in the range (0,1), 'black' is 0,0,0,1

If the color model of the internal object was `unitialized`, then it was initialized to the values representing 'black' in the colorspace `defaultcolormodel` that was in effect at the time of the `shipout`.

7.7.3 Transforms

Each transform is a six-item array.

1	number	represents x
2	number	represents y
3	number	represents xx
4	number	represents yx
5	number	represents xy
6	number	represents yy

Note that the translation (index 1 and 2) comes first. This differs from the ordering in PostScript, where the translation comes last.

7.7.4 Dashes

Each `dash` is two-item hash, using the same model as PostScript for the representation of the dashlist. `dashes` is an array of 'on' and 'off', values, and `offset` is the phase of the pattern.

dashes	hash	an array of on-off numbers
offset	number	the starting offset value

7.8 Character size information

These functions find the size of a glyph in a defined font. The `fontname` is the same name as the argument to `infont`; the `char` is a glyph id in the range 0 to 255; the returned `w` is in AFM units.

7.8.1 mp.char_width

```
<number> w = mp.char_width(<string> fontname, <number> char)
```

7.8.2 mp.char_height

```
<number> w = mp.char_height(<string> fontname, <number> char)
```

7.8.3 mp.char_depth

```
<number> w = mp.char_depth(<string> fontname, <number> char)
```

7.9 Solving path control points

```
<boolean> success = mp.solve_path(<table> knots, <boolean> cyclic)
```

This modifies the `knots` table (which should contain an array of points in a path, with the substructure explained below) by filling in the control points. The boolean `cyclic` is used to determine whether the path should be the equivalent of `--cycle`. If the return value is `false`, there is an extra return argument containing the error string.

On entry, the individual knot tables can contain the values mentioned above (but typically the `left_{x,y}` and `right_{x,y}` will be missing). `{x,y}_coord` are both required. Also, some extra values are allowed:

`left_tension`	number	A tension specifier
`right_tension`	number	like `left_tension`
`left_curl`	number	A curl specifier
`right_curl`	number	like `left_curl`
`direction_x`	number	x displacement of a direction specifier
`direction_y`	number	y displacement of a direction specifier

Note the following:

- A knot has either a direction specifier, or a curl specifier, or a tension specification, or explicit control points, with the note that tensions, curls and control points are split in a left and a right side.
- The absolute value of a tension specifier should be more than 0.75 and less than 4096.0, with negative values indicating '`atleast`'.
- The absolute value of a direction or curl should be less than 4096.0.

- If a tension, curl, or direction is specified, then existing control points will be replaced by the newly computed value.
- Calling `solve_path` does not effect the current mplib instance, but it does need a properly initialized instance to be able to function properly.