1. 1. Makempx overview.

This source file implements the makempx functionality for the new MetaPost. It includes all of the functional code from the old standalone programs

mpto
dmp
dvitomp
makempx

combined into one, with many changes to make all of the code cooperate nicely.

2. Header files

The local C preprocessor definitions have to come after the C includes in order to prevent name clashes.

```
#include <w2c/config.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <assert.h>
#include <setjmp.h>
#include <errno.h>      /* TODO autoconf ? */      /* unistd.h is needed for every non-Win32
    platform, and we assume * that implies that sys/types.h is also present */
#ifndef WIN32
#include <sys/types.h>
#include <unistd.h>
#endif      /* processes */
#ifdef WIN32
#include <io.h>
#include <process.h>
#else
#if HAVE_SYS_WAIT_H
#include <sys/wait.h>
#endif
#ifndef WEXITSTATUS
#define WEXITSTATUS(stat_val)  ((unsigned)(stat_val) >> 8)
#endif
#ifndef WIFEXITED
#define WIFEXITED(stat_val)  (((stat_val) & 255) == 0)
#endif
#endif      /* directories */
#ifdef WIN32
#include <direct.h>
#else
#if HAVE_DIRENT_H
#include <dirent.h>
#else
#define dirent  direct
#if HAVE_SYS_NDIR_H
#include <sys/ndir.h>
#endif
#if HAVE_SYS_DIR_H
#include <sys/dir.h>
#endif
#if HAVE_NDIR_H
#include <ndir.h>
#endif
#endif
#endif
#if HAVE_SYS_STAT_H
#include <sys/stat.h>
#endif
#include <ctype.h>
#include <time.h>
```

```
#include <math.h>
#define trunc(x) ((integer)(x))
#define fabs(x) ((x) < 0 ? (−(x)) : (x))
#define floor(x) ((integer)(fabs(x)))
#ifndef PI
#define PI 3.14159265358979323846
#endif
#include "avl.h"
#include "mpxout.h"
  ⟨ Preprocessor definitions ⟩
```

3. Data types

From the Pascal code of DVItoMP two implicit types are inherited: *web_boolean* and *web_integer*.

The more complex datatypes are defined in the following sections.

```
#define true 1
#define false 0
  typedef signed int web_integer;
  typedef signed int web_boolean; ⟨ C Data Types 5 ⟩⟨ Declarations 20 ⟩
```

4. The single most important data structure is the structure *mpx_data*. It contains all of the global state for a specific *makempx* run. A pointer to it is passed as the first argument to just about every function call.

One of the fields is a bit special because it is so important: *mode* is the decider between running TEX or Troff as the typesetting engine.

```
⟨ mpxout.h 4 ⟩ ≡
#ifndef MPXOUT_H
#define MPXOUT_H 1
  typedef enum {
    mpx_tex_mode = 0, mpx_troff_mode = 1
  } mpx_modes;
  typedef struct mpx_data *MPX; ⟨ Makempx header information 157 ⟩
#endif
```

5. ⟨ C Data Types 5 ⟩ ≡

```
  ⟨ Types in the outer block 8 ⟩
      typedef struct mpx_data {
        int mode;
        ⟨ Globals 9 ⟩
      } mpx_data;
```

This code is used in section 3.

6. Here are some macros for common programming idioms.

```
#define MAXINT #7FFFFF /* somewhat arbitrary */
#define incr(A) (A) = (A) + 1 /* increase a variable by unity */
#define decr(A) (A) = (A) − 1 /* decrease a variable by unity */
```

7. Once an MPX object is allocated, the memory it occupies needs to be initialized to a usable state. This procedure gets things started properly.

This function is not allowed to run *mpx_abort* because at this point the jump buffer is not yet initialized, so it should only be used for things that cannot go wrong!

```
static void mpx_initialize(MPX mpx)
{
  memset(mpx, 0, sizeof(struct mpx_data));
  ⟨ Set initial values 10 ⟩
}
```

8. A global variable *history* keeps track of what type of errors have occurred with the hope that that MetaPost can be warned of any problems.

```
⟨ Types in the outer block 8 ⟩ ≡
  enum mpx_history_states {
    mpx_spotless = 0,      /* history value when no problems have been found */
    mpx_cksum_trouble,     /* history value there have been font checksum mismatches */
    mpx_warning_given,     /* history value after a recoverable error */
    mpx_fatal_error        /* history value if processing had to be aborted */
  };
```

See also sections 131, 165, and 190.

This code is used in section 5.

9.

```
⟨ Globals 9 ⟩ ≡
  int history;
```

See also sections 11, 16, 23, 40, 44, 45, 47, 55, 63, 67, 87, 93, 95, 107, 111, 124, 128, 132, 142, 155, 158, 169, 174, 179, 182, 189, 197, 210, and 222.

This code is used in section 5.

10.

```
⟨ Set initial values 10 ⟩ ≡
  mpx→history = mpx_spotless;
```

See also sections 24, 48, 56, 88, 92, 135, 143, 156, and 161.

This code is used in section 7.

11. The structure has room for the names and the **FILE** * for the input and output files. The most important ones are listed here, the variables for the intermediate files are declared where they are needed.

```
⟨ Globals 9 ⟩ +≡
  char *banner;
  char *mpname;
  FILE *mpfile;
  char *mpxname;
  FILE *mpxfile;
  FILE *errfile;
  int lnno;      /* current line number */
```

12. A set of basic reporting functions.

```
static void mpx_printf(MPX mpx, const char *header, const char *msg, va_list ap)
{
  fprintf(mpx→errfile, "makempx␣%s:␣%s:", header, mpx→mpname);
  if (mpx→lnno ≠ 0) fprintf(mpx→errfile, "%d:", mpx→lnno);
  fprintf(mpx→errfile, "␣");
  (void) vfprintf(mpx→errfile, msg, ap);
  fprintf(mpx→errfile, "\n");
}
```

13.

```
static void mpx_report(MPX mpx, const char *msg, ...)
{
  va_list ap;
  if (mpx→debug ≡ 0) return;
  va_start(ap, msg);
  mpx_printf(mpx, "debug", msg, ap);
  va_end(ap);
  if (mpx→history < mpx_warning_given) mpx→history = mpx_cksum_trouble;
}
```

14.

```
static void mpx_warn(MPX mpx, const char *msg, ...)
{
  va_list ap;
  va_start(ap, msg);
  mpx_printf(mpx, "warning", msg, ap);
  va_end(ap);
  if (mpx→history < mpx_warning_given) mpx→history = mpx_cksum_trouble;
}
```

15.

```
static void mpx_error(MPX mpx, const char *msg, ...)
{
  va_list ap;
  va_start(ap, msg);
  mpx_printf(mpx, "error", msg, ap);
  va_end(ap);
  mpx→history = mpx_warning_given;
}
```

16. The program uses a *jump_buf* to handle non-local returns, this is initialized at a single spot: the start of *mp_makempx*.

```
#define mpx_jump_out longjmp(mpx→jump_buf, 1)
⟨ Globals 9 ⟩ +≡
  jmp_buf jump_buf;
```

17.

```
static void mpx_abort(MPX mpx, const char *msg, ...)
{
  va_list ap;
  va_start(ap, msg);
  fprintf(stderr, "fatal: ");
  (void) vfprintf(stderr, msg, ap);
  va_end(ap);
  va_start(ap, msg);
  mpx_printf(mpx, "fatal", msg, ap);
  va_end(ap);
  mpx->history = mpx_fatal_error;
  mpx_erasetmp(mpx);
  mpx_jump_out;
}
```

18. ⟨ Install and test the non-local jump buffer 18 ⟩ ≡

```
if (setjmp(mpx->jump_buf) ≠ 0) {
  int h = mpx->history;
  xfree(mpx->buf);
  xfree(mpx->maincmd);
  xfree(mpx->mpname);
  xfree(mpx->mpxname);
  xfree(mpx);
  return h;
}
```

This code is used in section 226.

19.

```
static FILE *mpx_xfopen(MPX mpx, const char *fname, const char *fmode)
{
  FILE *f = fopen(fname, fmode);
  if (f ≡ Λ) mpx_abort(mpx, "File open error for %s in mode %s", fname, fmode);
  return f;
}
static void mpx_fclose(MPX mpx, FILE *file)
{
  (void) mpx;
  (void) fclose(file);
}
```

20.

```
#define xfree(A) do
        {
          mpx_xfree(A);
          A = Λ;
        }
        while (0)
#define xrealloc(P, A, B) mpx_xrealloc(mpx, P, A, B)
#define xmalloc(A, B) mpx_xmalloc(mpx, A, B)
#define xstrdup(A) mpx_xstrdup(mpx, A)
```

⟨ Declarations 20 ⟩ ≡

```
  static void mpx_xfree(void *x);
  static void *mpx_xrealloc(MPX mpx, void *p, size_t nmem, size_t size);
  static void *mpx_xmalloc(MPX mpx, size_t nmem, size_t size);
  static char *mpx_xstrdup(MPX mpX, const char *s);
```

See also sections 96, 100, 134, 159, 162, 183, and 212.

This code is used in section 3.

21. The *max_size_test* guards against overflow, on the assumption that **size_t** is at least 31bits wide.

```
#define max_size_test #7FFFFFFF
  static void mpx_xfree(void *x)
  {
    if (x ≠ Λ) free(x);
  }
  static void *mpx_xrealloc(MPX mpx, void *p, size_t nmem, size_t size)
  {
    void *w;
    if ((max_size_test/size) < nmem) {
      mpx_abort(mpx, "Memory␣size␣overflow");
    }
    w = realloc(p, (nmem * size));
    if (w ≡ Λ) mpx_abort(mpx, "Out␣of␣Memory");
    return w;
  }
  static void *mpx_xmalloc(MPX mpx, size_t nmem, size_t size)
  {
    void *w;
    if ((max_size_test/size) < nmem) {
      mpx_abort(mpx, "Memory␣size␣overflow");
    }
    w = malloc(nmem * size);
    if (w ≡ Λ) mpx_abort(mpx, "Out␣of␣Memory");
    return w;
  }
  static char *mpx_xstrdup(MPX mpx, const char *s)
  {
    char *w;
    if (s ≡ Λ) return Λ;
    w = strdup(s);
    if (w ≡ Λ) mpx_abort(mpx, "Out␣of␣Memory");
    return w;
  }
```

22. The command 'newer' became a function.

We may have high-res timers in struct stat. If we do, use them.

```
static int mpx_newer(char *source, char *target)
{
  struct stat source_stat, target_stat;
#if HAVE_SYS_STAT_H
  if (stat(target, &target_stat) < 0) return 0;      /* true */
  if (stat(source, &source_stat) < 0) return 1;      /* false */
#if HAVE_STRUCT_STAT_ST_MTIM
  if (source_stat.st_mtim.tv_sec > target_stat.st_mtim.tv_sec ∨ (source_stat.st_mtim.tv_sec ≡
        target_stat.st_mtim.tv_sec ∧ source_stat.st_mtim.tv_nsec ≥ target_stat.st_mtim.tv_nsec))
    return 0;
#else
  if (source_stat.st_mtime ≥ target_stat.st_mtime) return 0;
#endif
#endif
  return 1;
}
```

23. Extracting data from MetaPost input.

This part of the program transforms a MetaPost input file into a TEX or troff input file by stripping out `btex...etex` and `verbatimtex...etex` sections. Leading and trailing spaces and tabs are removed from the extracted material and it is surrounded by the preceding and following strings defined immediately below. The input file should be given as argument 1 and the resulting TEX or troff file is written on standard output.

John Hobby wrote the original version, which has since been extensively altered. The current implementation is a bit trickier than I would like, but changing it will take careful study and will likely make it run slower, so I've left it as-is for now.

```
⟨ Globals 9 ⟩ +≡
  int texcnt;      /* btex..etex blocks so far */
  int verbcnt;     /* verbatimtex..etex blocks so far */
  char *bb, *tt, *aa;      /* start of before, token, and after strings */
  char *buf;      /* the input line */
  unsigned bufsize;
```

24. ⟨ Set initial values 10 ⟩ +≡

```
  mpx->bufsize = 1000;
```

25. This function returns NULL on EOF, otherwise it returns *buf*.

```
static char *mpx_getline(MPX mpx, FILE *mpfile)
{
  int c;
  unsigned loc = 0;
  if (feof(mpfile)) return Λ;
  if (mpx->buf ≡ Λ) mpx->buf = xmalloc(mpx->bufsize, 1);
  while ((c = getc(mpfile)) ≠ EOF ∧ c ≠ '\n' ∧ c ≠ '\r') {
    mpx->buf[loc++] = (char) c;
    if (loc ≡ mpx->bufsize) {
      char *temp = mpx->buf;
      unsigned n = mpx->bufsize + (mpx->bufsize ≫ 4);
      if (n > MAXINT) mpx_abort(mpx, "Line is too long");
      mpx->buf = xmalloc(n, 1);
      memcpy(mpx->buf, temp, mpx->bufsize);
      free(temp);
      mpx->bufsize = n;
    }
  }
  mpx->buf[loc] = 0;
  if (c ≡ '\r') {
    c = getc(mpfile);
    if (c ≠ '\n') ungetc(c, mpfile);
  }
  mpx->lnno++;
  return mpx->buf;
}
```

26. Return nonzero if a prefix of string s matches the null-terminated string t and the next character is not a letter or an underscore.

```
static int mpx_match_str(const char *s, const char *t)
{
  while (*t ≠ 0) {
    if (*s ≠ *t) return 0;
    s++;
    t++;
  }
  if ((*s ≥ 'a' ∧ *s ≤ 'z') ∨ (*s ≥ 'A' ∧ *s ≤ 'Z') ∨ *s ≡ '_') return 0;
  return 1;
}
```

27. This function tries to express s as the concatenation of three strings b, t, a, with the global pointers bb, tt, and aa set to the start of the corresponding strings. String t is either a quote mark, a percent sign, or an alphabetic token `btex`, `etex`, or `verbatimtex`. (An alphabetic token is a maximal sequence of letters and underscores.) If there are several possible substrings t, we choose the leftmost one. If there is no such t, we set $b = s$ and return 0.

Various values are defined, so that *mpx_copy_mpto* can distinguish between `verbatimtex` ... `etex` and `btex` ... `etex` (the former has no whitespace corrections applied).

```
#define VERBATIM_TEX 1
#define B_TEX 2
#define FIRST_VERBATIM_TEX 3
  static int mpx_getbta(MPX mpx, char *s)
  {
    int ok = 1;      /* zero if last character was a - z, A - Z, or _ */
    mpx->bb = s;
    if (s == Λ) {
      mpx->tt = Λ;
      mpx->aa = Λ;
      return 0;
    }
    for (mpx->tt = mpx->bb; *(mpx->tt) ≠ 0; mpx->tt++) {
      switch (*(mpx->tt)) {
      case '"': case '%': mpx->aa = mpx->tt + 1;
        return 1;
      case 'b':
        if (ok ∧ mpx_match_str(mpx->tt, "btex")) {
          mpx->aa = mpx->tt + 4;
          return 1;
        }
        else {
          ok = 0;
        }
        break;
      case 'e':
        if (ok ∧ mpx_match_str(mpx->tt, "etex")) {
          mpx->aa = mpx->tt + 4;
          return 1;
        }
        else {
          ok = 0;
        }
        break;
      case 'v':
        if (ok ∧ mpx_match_str(mpx->tt, "verbatimtex")) {
          mpx->aa = mpx->tt + 11;
          return 1;
        }
        else {
          ok = 0;
        }
        break;
      default:
```

```
      if ((*(mpx→tt) ≥ 'a' ∧ *(mpx→tt) ≤ 'z') ∨ (*(mpx→tt) ≥ 'A' ∧ *(mpx→tt) ≤ 'Z') ∨ (*(mpx→tt) ≡ '_'))
        ok = 0;
      else ok = 1;
    }
  }
  mpx→aa = mpx→tt;
  return 0;
}
```

```
28. static void mpx_copy_mpto(MPX mpx, FILE *outfile, int textype)
{
  char *s;      /* where a string to print stops */
  char *t;      /* for finding start of last line */
  char c;
  char *res = Λ;
  t = Λ;
  do {
    if (mpx→aa ≡ Λ ∨ *mpx→aa ≡ 0) {
      if ((mpx→aa = mpx_getline(mpx, mpx→mpfile)) ≡ Λ) {
        mpx_error(mpx, "btex␣section␣does␣not␣end");
        return;
      }
    }
    if (mpx_getbta(mpx, mpx→aa) ∧ *(mpx→tt) ≡ 'e') {
      s = mpx→tt;
    }
    else {
      if (mpx→tt ≡ Λ) {
        mpx_error(mpx, "btex␣section␣does␣not␣end");
        return;
      }
      else if (*(mpx→tt) ≡ 'b') {
        mpx_error(mpx, "btex␣in␣TeX␣mode");
        return;
      }
      else if (*(mpx→tt) ≡ 'v') {
        mpx_error(mpx, "verbatimtex␣in␣TeX␣mode");
        return;
      }
      s = mpx→aa;
    }
    c = *s;
    *s = 0;
    if (res ≡ Λ) {
      res = xmalloc(strlen(mpx→bb) + 2, 1);
      res = strncpy(res, mpx→bb, (strlen(mpx→bb) + 1));
    }
    else {
      res = xrealloc(res, strlen(res) + strlen(mpx→bb) + 2, 1);
      res = strncat(res, mpx→bb, strlen(mpx→bb));
    }
    if (c ≡ '\0') res = strncat(res, "\n", 1);
    *s = c;
  } while (*(mpx→tt) ≠ 'e');
  s = res;
  if (textype ≡ B_TEX) {      /* whitespace at the end */
    for (s = res + strlen(res) − 1; s ≥ res ∧ (*s ≡ '␣' ∨ *s ≡ '\t' ∨ *s ≡ '\r' ∨ *s ≡ '\n'); s−−) ;
    t = s;
    *(++s) = '\0';
  }
  else {
```

```
    t = s;
  }
  if (textype ≡ B_TEX ∨ textype ≡ FIRST_VERBATIM_TEX) {      /* whitespace at the start */
    for (s = res; s < (res + strlen(res)) ∧ (*s ≡ '␣' ∨ *s ≡ '\t' ∨ *s ≡ '\r' ∨ *s ≡ '\n'); s++) ;
    for ( ; *t ≠ '\n' ∧ t > s; t--) ;
  }
  fprintf(outfile, "%s", s);
  if (textype ≡ B_TEX) {      /* put no % at end if it's only 1 line total, starting with %; * this covers
        the special case % &format in a single line. */
    if (t ≠ s ∨ *t ≠ '%') fprintf(outfile, "%%");
  }
  free(res);
}
```

29. Static strings for mpto

```
static const char *mpx_predoc[] = {"", ".po␣0\n"};
static const char *mpx_postdoc[] = {"\\end{document}\n", ""};
static const char *mpx_pretex1[] = {"\\gdef\\mpxshipout{\\shipout\\hbox\\bgr\
    oup%\n""␣␣\\setbox0=\\hbox\\bgroup}%\n""\\gdef\\stopmpxship\
    out{\\egroup""␣␣\\dimen0=\\ht0␣\\advance\\dimen0\\dp0\n""␣␣\\dimen1=\\ht0␣\\\
    dimen2=\\dp0\n""␣␣\\setbox0=\\hbox\\bgroup\n""␣␣␣␣\\box0\n""␣␣␣␣\\ifnum\\dimen0\
    >0␣\\vrule␣width1sp␣height\\dimen1␣depth\\dimen2␣\n""␣␣␣␣\\else␣\\vrule␣\
    width1sp␣height1sp␣depth0sp\\relax\n""␣␣␣␣\\fi\\egroup\n""␣␣\\ht0=0pt␣\\dp0=0\
    pt␣\\box0␣\\egroup}\n""\\mpxshipout%%␣line␣%d␣%s\n", ".lf␣%d␣%s\n"};
static const char *mpx_pretex[] = {"\\mpxshipout%%␣line␣%d␣%s\n", ".bp\n.lf␣%d␣%s\n"};
static const char *mpx_posttex[] = {"\n\\stopmpxshipout\n", "\n"};
static const char *mpx_preverb1[] = {"", ".lf␣%d␣%s\n"};      /* if very first instance */
static const char *mpx_preverb[] = {"%%␣line␣%d␣%s\n", ".lf␣%d␣%s\n"};
  /* all other instances */
static const char *mpx_postverb[] = {"\n", "\n"};
```

```
30. static void mpx_mpto(MPX mpx, char *tmpname, char *mptexpre)
{
  FILE *outfile;
  int verbatim_written = 0;
  int mode = mpx→mode;
  char *mpname = mpx→mpname;
  if (mode ≡ mpx_tex_mode) {
    TMPNAME_EXT(mpx→tex, ".tex");
  }
  else {
    TMPNAME_EXT(mpx→tex, ".i");
  }
  outfile = mpx_xfopen(mpx, mpx→tex, "wb");
  if (mode ≡ mpx_tex_mode) {
    FILE *fr;
    if ((fr = fopen(mptexpre, "r")) ≠ Λ) {
      size_t i;
      char buf[512];
      while ((i = fread((void *) buf, 1, 512, fr)) > 0) {
        fwrite((void *) buf, 1, i, outfile);
      }
      mpx_fclose(mpx, fr);
    }
  }
  mpx→mpfile = mpx_xfopen(mpx, mpname, "r");
  fprintf(outfile, "%s", mpx_predoc[mode]);
  while (mpx_getline(mpx, mpx→mpfile) ≠ Λ) ⟨ Do a line 31 ⟩;
  fprintf(outfile, "%s", mpx_postdoc[mode]);
  mpx_fclose(mpx, mpx→mpfile);
  mpx_fclose(mpx, outfile);
  mpx→lnno = 0;
}
```

31.

```
⟨ Do a line 31 ⟩ ≡
  {
    mpx→aa = mpx→buf;
    while (mpx_getbta(mpx, mpx→aa)) {
      if (*(mpx→tt) ≡ '%') {
        break;
      }
      else if (*(mpx→tt) ≡ '"') {
        do {
          if (¬mpx_getbta(mpx, mpx→aa)) mpx_error(mpx, "string␣does␣not␣end");
        } while (*(mpx→tt) ≠ '"');
      }
      else if (*(mpx→tt) ≡ 'b') {
        if (mpx→texcnt++ ≡ 0) fprintf(outfile, mpx_pretex1[mode], mpx→lnno, mpname);
        else fprintf(outfile, mpx_pretex[mode], mpx→lnno, mpname);
        mpx_copy_mpto(mpx, outfile, B_TEX);
        fprintf(outfile, "%s", mpx_posttex[mode]);
      }
      else if (*(mpx→tt) ≡ 'v') {
        if (mpx→verbcnt++ ≡ 0 ∧ mpx→texcnt ≡ 0)
          fprintf(outfile, mpx_preverb1[mode], mpx→lnno, mpname);
        else fprintf(outfile, mpx_preverb[mode], mpx→lnno, mpname);
        if (¬verbatim_written) mpx_copy_mpto(mpx, outfile, FIRST_VERBATIM_TEX);
        else mpx_copy_mpto(mpx, outfile, VERBATIM_TEX);
        fprintf(outfile, "%s", mpx_postverb[mode]);
      }
      else {
        mpx_error(mpx, "unmatched␣etex");
      }
      verbatim_written = 1;
    }
  }
```

This code is used in section 30.

32. ⟨ Run *mpto* on the mp file 32 ⟩ ≡

```
mpx_mpto(mpx, tmpname, mpxopt→mptexpre)
```

This code is used in section 226.

33. DVItoMP Processing.

The `DVItoMP` program reads binary device-independent ("`DVI`") files that are produced by document compilers such as TEX, and converts them into a symbolic form understood by MetaPost. It is loosely based on the `DVItype` utility program that produces a more faithful symbolic form of a `DVI` file.

The output file is a sequence of MetaPost picture expressions, one for every page in the `DVI` file. It makes no difference to `DVItoMP` where the `DVI` file comes from, but it is intended to process the result of running TEX or LATEX on the output of the extraction process that is defined above. Such a `DVI` file will contain one page for every `btex...etex` block in the original input. Processing with `DVItoMP` creates a corresponding sequence of MetaPost picture expressions for use as an auxiliary input file. Since MetaPost expects such files to have the extension `.MPX`, the output of `DVItoMP` is sometimes called an "`MPX`" file.

34. The following parameters can be changed at compile time to extend or reduce `DVItoMP`'s capacity.

TODO: dynamic reallocation

```
#define virtual_space 1000000 /* maximum total bytes of typesetting commands for virtual fonts */
#define max_fonts 1000 /* maximum number of distinct fonts per DVI file */
#define max_fnums 3000 /* maximum number of fonts plus fonts local to virtual fonts */
#define max_widths (256 * max_fonts)
  /* maximum number of different characters among all fonts */
#define line_length 79 /* maximum output line length (must be at least 60) */
#define stack_size 100 /* DVI files shouldn't push beyond this depth */
#define font_tolerance 0.00001 /* font sizes should match to within this multiple of 2^20 DVI units */
```

35. If the `DVI` file is badly malformed, the whole process must be aborted; `DVItoMP` will give up, after issuing an error message about the symptoms that were noticed.

```
#define bad_dvi(A) mpx_abort(mpx, "Bad␣DVI␣file:␣"A"!")
#define bad_dvi_two(A,B) mpx_abort(mpx, "Bad␣DVI␣file:␣"A"!", B)
```

36. The character set.

Like all programs written with the `WEB` system, `DVItoMP` can be used with any character set. It an identify transfrom internally, because the programming for portable input-output is easier when a fixed internal code is used, and because `DVI` files use ASCII code for file names.

In the conversion from Pascal to C, the *xchr* array has been removed. Because some systems may still want to change the input–output character set, the accesses to *xchr* and *printable* are replaced by macro calls.

#define *printable*(*c*) (*isprint*(*c*) ∧ *c* < 128 ∧ *c* ≠ '"')
#define *xchr*(*A*) (*A*)

37. **static void** *mpx_open_mpxfile*(**MPX** *mpx*)
{ /* prepares to write text on *mpxfile* */
mpx→*mpxfile* = *mpx_xfopen*(*mpx*, *mpx*→*mpxname*, "wb");
}

38. Device-independent file format. The format of `DVI` files is described in many places including `dvitype.web` and Volume B of D. E. Knuth's *Computers and Typesetting*. This program refers to the following command codes.

```
#define id_byte 2 /* identifies the kind of DVI files described here */
#define set_char_0 0 /* typeset character 0 and move right */
#define set1 128 /* typeset a character and move right */
#define set_rule 132 /* typeset a rule and move right */
#define put1 133 /* typeset a character */
#define put_rule 137 /* typeset a rule */
#define nop 138 /* no operation */
#define bop 139 /* beginning of page */
#define eop 140 /* ending of page */
#define push 141 /* save the current positions */
#define pop 142 /* restore previous positions */
#define right1 143 /* move right */
#define w0 147 /* move right by w */
#define w1 148 /* move right and set w */
#define x0 152 /* move right by x */
#define x1 153 /* move right and set x */
#define down1 157 /* move down */
#define y0 161 /* move down by y */
#define y1 162 /* move down and set y */
#define z0 166 /* move down by z */
#define z1 167 /* move down and set z */
#define fnt_num_0 171 /* set current font to 0 */
#define fnt1 235 /* set current font */
#define xxx1 239 /* extension to DVI primitives */
#define xxx4 242 /* potentially long extension to DVI primitives */
#define fnt_def1 243 /* define the meaning of a font number */
#define pre 247 /* preamble */
#define post 248 /* postamble beginning */
#define post_post 249 /* postamble ending */
#define undefined_commands 250: case 251: case 252: case 253: case 254: case 255
```

39. Input from binary files.

40. The program deals with two binary file variables: *dvi_file* is the main input file that we are translating into symbolic form, and *tfm_file* is the current font metric file from which character-width information is being read. It is convenient to have a throw-away variable for function results when reading parts of the files that are being skipped.

```
⟨ Globals 9 ⟩ +≡
  FILE *dvi_file;      /* the input file */
  FILE *tfm_file;      /* a font metric file */
  FILE *vf_file;       /* a virtual font file */
```

41. Prepares to read packed bytes in *dvi_file*

```
  static void mpx_open_dvi_file(MPX mpx)
  {
    mpx→dvi_file = fopen(mpx→dviname, "rb");
    if (mpx→dvi_file ≡ Λ) mpx_abort(mpx, "DVI␣generation␣failed");
  }
```

42. Prepares to read packed bytes in *tfm_file*

```
  static web_boolean mpx_open_tfm_file(MPX mpx)
  {
    mpx→tfm_file = mpx_fsearch(mpx, mpx→cur_name, mpx_tfm_format);
    if (mpx→tfm_file ≡ Λ) mpx_abort(mpx, "Cannot␣find␣TFM␣%s", mpx→cur_name);
    free(mpx→cur_name);      /* We xmalloc'd this before we got called. */
    return true;      /* If we get here, we succeeded. */
  }
```

43. Prepares to read packed bytes in *vf_file*. It's ok if the `VF` file doesn't exist.

```
  static web_boolean mpx_open_vf_file(MPX mpx)
  {
    mpx→vf_file = mpx_fsearch(mpx, mpx→cur_name, mpx_vf_format);
    if (mpx→vf_file) {
      free(mpx→cur_name);
      return true;
    }
    return false;
  }
```

44. If you looked carefully at the preceding code, you probably asked, "What is *cur_name*?" Good question. It's a global variable: *cur_name* is a string variable that will be set to the current font metric file name before *open_tfm_file* or *open_vf_file* is called.

```
⟨ Globals 9 ⟩ +≡
  char *cur_name;      /* external name */
```

45. It turns out to be convenient to read four bytes at a time, when we are inputting from `TFM` files. The input goes into global variables *b0*, *b1*, *b2*, and *b3*, with *b0* getting the first byte and *b3* the fourth.

```
⟨ Globals 9 ⟩ +≡
  int b0, b1, b2, b3;      /* four bytes input at once */
```

46. The *read_tfm_word* procedure sets *b0* through *b3* to the next four bytes in the current `TFM` file.

```
static void mpx_read_tfm_word(MPX mpx)
{
  mpx→b0 = getc(mpx→tfm_file);
  mpx→b1 = getc(mpx→tfm_file);
  mpx→b2 = getc(mpx→tfm_file);
  mpx→b3 = getc(mpx→tfm_file);
}
```

47. Input can come from from three different sources depending on the settings of global variables. When *vf_reading* is true, we read from the `VF` file. Otherwise, input can either come directly from *dvi_file* or from a buffer *cmd_buf*. The latter case applies whenever *buf_ptr* < *virtual_space*.

```
⟨ Globals 9 ⟩ +≡
  web_boolean vf_reading;      /* should input come from vf_file? */
  unsigned char cmd_buf[(virtual_space + 1)];      /* commands for virtual characters */
  unsigned int buf_ptr;      /* cmd_buf index for the next byte */
```

48.
```
⟨ Set initial values 10 ⟩ +≡
  mpx→vf_reading = false;
  mpx→buf_ptr = virtual_space;
```

49. We shall use a set of simple functions to read the next byte or bytes from the current input source. There are seven possibilities, each of which is treated as a separate function in order to minimize the overhead for subroutine calls.

```
static web_integer mpx_get_byte(MPX mpx)
{   /* returns the next byte, unsigned */
  unsigned char b;
  ⟨ Read one byte into b 50 ⟩;
  return b;
}
static web_integer mpx_signed_byte(MPX mpx)
{   /* returns the next byte, signed */
  unsigned char b;
  ⟨ Read one byte into b 50 ⟩;
  return (b < 128 ? b : (b − 256));
}
static web_integer mpx_get_two_bytes(MPX mpx)
{   /* returns the next two bytes, unsigned */
  unsigned char a, b;
  a = 0;
  b = 0;   /* for compiler warnings */
  ⟨ Read two bytes into a and b 51 ⟩;
  return (a * (int)(256) + b);
}
static web_integer mpx_signed_pair(MPX mpx)
{   /* returns the next two bytes, signed */
  unsigned char a, b;
  a = 0;
  b = 0;   /* for compiler warnings */
  ⟨ Read two bytes into a and b 51 ⟩;
  if (a < 128) return (a * 256 + b);
  else return ((a − 256) * 256 + b);
}
static web_integer mpx_get_three_bytes(MPX mpx)
{   /* returns the next three bytes, unsigned */
  unsigned char a, b, c;
  a = 0;
  b = 0;
  c = 0;   /* for compiler warnings */
  ⟨ Read three bytes into a, b, and c 52 ⟩;
  return ((a * (int)(256) + b) * 256 + c);
}
static web_integer mpx_signed_trio(MPX mpx)
{   /* returns the next three bytes, signed */
  unsigned char a, b, c;
  a = 0;
  b = 0;
  c = 0;   /* for compiler warnings */
  ⟨ Read three bytes into a, b, and c 52 ⟩;
  if (a < 128) return ((a * (int)(256) + b) * 256 + c);
```

```
  else return (((a − (int)(256)) ∗ 256 + b) ∗ 256 + c);
}
static web_integer mpx_signed_quad(MPX mpx)
{   /∗ returns the next four bytes, signed ∗/
  unsigned char a, b, c, d;
  a = 0;
  b = 0;
  c = 0;
  d = 0;   /∗ for compiler warnings ∗/
  ⟨ Read four bytes into a, b, c, and d 53 ⟩;
  if (a < 128) return (((a ∗ (int)(256) + b) ∗ 256 + c) ∗ 256 + d);
  else return ((((a − 256) ∗ (int)(256) + b) ∗ 256 + c) ∗ 256 + d);
}
```

50. ⟨ Read one byte into b 50 ⟩ ≡

```
if (mpx→vf_reading) {
  b = (unsigned char) getc(mpx→vf_file);
}
else if (mpx→buf_ptr ≡ virtual_space) {
  b = (unsigned char) getc(mpx→dvi_file);
}
else {
  b = mpx→cmd_buf[mpx→buf_ptr];
  incr(mpx→buf_ptr);
}
```

This code is used in section 49.

51. ⟨ Read two bytes into a and b 51 ⟩ ≡

```
if (mpx→vf_reading) {
  a = (unsigned char) getc(mpx→vf_file);
  b = (unsigned char) getc(mpx→vf_file);
}
else if (mpx→buf_ptr ≡ virtual_space) {
  a = (unsigned char) getc(mpx→dvi_file);
  b = (unsigned char) getc(mpx→dvi_file);
}
else if (mpx→buf_ptr + 2 > mpx→n_cmds) {
  mpx_abort(mpx, "Error␣detected␣while␣interpreting␣a␣virtual␣font");
}
else {
  a = mpx→cmd_buf[mpx→buf_ptr];
  b = mpx→cmd_buf[mpx→buf_ptr + 1];
  mpx→buf_ptr += 2;
}
```

This code is used in section 49.

52. ⟨ Read three bytes into a, b, and c 52 ⟩ ≡

```
if (mpx→vf_reading) {
  a = (unsigned char) getc(mpx→vf_file);
  b = (unsigned char) getc(mpx→vf_file);
  c = (unsigned char) getc(mpx→vf_file);
}
else if (mpx→buf_ptr ≡ virtual_space) {
  a = (unsigned char) getc(mpx→dvi_file);
  b = (unsigned char) getc(mpx→dvi_file);
  c = (unsigned char) getc(mpx→dvi_file);
}
else if (mpx→buf_ptr + 3 > mpx→n_cmds) {
  mpx_abort(mpx, "Error␣detected␣while␣interpreting␣a␣virtual␣font");
}
else {
  a = mpx→cmd_buf[mpx→buf_ptr];
  b = mpx→cmd_buf[mpx→buf_ptr + 1];
  c = mpx→cmd_buf[mpx→buf_ptr + 2];
  mpx→buf_ptr += 3;
}
```

This code is used in section 49.

53. ⟨ Read four bytes into a, b, c, and d 53 ⟩ ≡

```
if (mpx→vf_reading) {
  a = (unsigned char) getc(mpx→vf_file);
  b = (unsigned char) getc(mpx→vf_file);
  c = (unsigned char) getc(mpx→vf_file);
  d = (unsigned char) getc(mpx→vf_file);
}
else if (mpx→buf_ptr ≡ virtual_space) {
  a = (unsigned char) getc(mpx→dvi_file);
  b = (unsigned char) getc(mpx→dvi_file);
  c = (unsigned char) getc(mpx→dvi_file);
  d = (unsigned char) getc(mpx→dvi_file);
}
else if (mpx→buf_ptr + 4 > mpx→n_cmds) {
  mpx_abort(mpx, "Error␣detected␣while␣interpreting␣a␣virtual␣font");
}
else {
  a = mpx→cmd_buf[mpx→buf_ptr];
  b = mpx→cmd_buf[mpx→buf_ptr + 1];
  c = mpx→cmd_buf[mpx→buf_ptr + 2];
  d = mpx→cmd_buf[mpx→buf_ptr + 3];
  mpx→buf_ptr += 4;
}
```

This code is used in section 49.

54. Data structures for fonts.

DVI file format does not include information about character widths, since that would tend to make the files a lot longer. But a program that reads a DVI file is supposed to know the widths of the characters that appear in *set_char* commands. Therefore DVItoMP looks at the font metric (TFM) files for the fonts that are involved.

55. For purposes of this program, the only thing we need to know about a given character c in a non-virtual font f is the width. For the font as a whole, all we need is the symbolic name to use in the MPX file.

This information appears implicitly in the following data structures. The current number of fonts defined is nf. Each such font has an internal number f, where $0 \le f < nf$. There is also an external number that identifies the font in the DVI file. The correspondence is maintained in arrays *font_num* and *internal_num* so that $font\_num[i]$ is the external number for $f = internal\_num[i]$. The external name of this font is the string that occupies $font\_name[f]$. The legal characters run from $font\_bc[f]$ to $font\_ec[f]$, inclusive. The TFM file can specify that some of these are invalid, but this doesn't concern DVItoMP because it does not do extensive error checking. The width of character c in font f is given by $char\_width(f,c) = width[info\_base[f]+c]$, and *info_ptr* is the first unused position of the *width* array.

If font f is a virtual font, there is a list of DVI commands for each character. These occupy consecutive positions in the *cmd_buf* array with the commands for character c starting at $start\_cmd(f,c) = cmd\_ptr[info\_base[f]+c]$ and ending just before $start\_cmd(f,c+1)$. Font numbers used when interpreting these DVI commands occupy positions $fbase[f]$ through $ftop[f]-1$ in the *font_num* table and the *internal_num* array gives the corresponding internal font numbers. If such an internal font number i does not correspond to some font occuring in the DVI file, then $font\_num[i]$ has not been assigned a meaningful value; this is indicated by $local\_only[i] = true$.

If font f is not virtual, then $fbase[f] = 0$ and $ftop[f] = 0$. The *start_cmd* values are ignored in this case.

```
#define char_width(A,B) mpx→width[mpx→info_base[(A)]+(B)]
#define start_cmd(A,B) mpx→cmd_ptr[mpx→info_base[(A)]+(B)]
⟨ Globals 9 ⟩ +≡
  web_integer font_num[(max_fnums+1)]; /* external font numbers */
  web_integer internal_num[(max_fnums+1)]; /* internal font numbers */
  web_boolean local_only[(max_fnums+1)]; /* font_num meaningless? */
  char *font_name[(max_fonts+1)]; /* starting positions of external font names */
  double font_scaled_size[(max_fonts+1)]; /* scale factors over 2^20 */
  double font_design_size[(max_fonts+1)]; /* design sizes over 2^20 */
  web_integer font_check_sum[(max_fonts+1)]; /* check sum from the font_def */
  web_integer font_bc[(max_fonts+1)]; /* beginning characters in fonts */
  web_integer font_ec[(max_fonts+1)]; /* ending characters in fonts */
  web_integer info_base[(max_fonts+1)]; /* index into width and cmd_ptr tables */
  web_integer width[(max_widths+1)]; /* character widths, in units 2^-20 of design size */
  web_integer fbase[(max_fonts+1)]; /* index into font_num for local fonts */
  web_integer ftop[(max_fonts+1)]; /* font_num index where local fonts stop */
  web_integer cmd_ptr[(max_widths+1)]; /* starting positions in cmd_buf */
  unsigned int nfonts; /* the number of known fonts */
  unsigned int vf_ptr; /* next font_num entry for virtual font font tables */
  unsigned int info_ptr; /* allocation pointer for width and cmd_ptr tables */
  unsigned int n_cmds; /* number of occupied cells in cmd_buf */
  unsigned int cur_fbase, cur_ftop; /* currently applicable part of the font_num table */
```

56. ⟨ Set initial values 10 ⟩ +≡

```
mpx→nfonts = 0;
mpx→info_ptr = 0;
mpx→font_name[0] = 0;
mpx→vf_ptr = max_fnums;
mpx→cur_fbase = 0;
mpx→cur_ftop = 0;
```

57. Printing the name of a given font is easy except that a procedure *print_char* is needed to actually send an *ASCII_code* to the MPX file.

⟨ Declare subroutines for printing strings 89 ⟩

```
static void mpx_print_font(MPX mpx, web_integer f)
{   /* f is an internal font number */
  if ((f < 0) ∨ (f ≥ (int) mpx→nfonts)) {
    bad_dvi("Undefined␣font");
  }
  else {
    char *s = mpx→font_name[f];
    while (*s) {
      mpx_print_char(mpx, (unsigned char) *s);
      s++;
    }
  }
}
```

58. Sometimes a font name is needed as part of an error message.

```
#define font_warn(A, B) mpx_warn(mpx, "%s␣%s", A, mpx→font_name[(B)])
#define font_error(A, B) mpx_error(mpx, "%s␣%s", A, mpx→font_name[(B)])
#define font_abort(A, B) mpx_abort(mpx, "%s␣%s", A, mpx→font_name[(B)])
```

59. When we encounter a font definition, we save the name, checksum, and size information, but we don't actually read the `TFM` or `VF` file until we are about to use the font. If a matching font is not already defined, we then allocate a new internal font number.

The following subroutine does the necessary things when a *fnt_def* command is encountered in the `DVI` file or in a `VF` file. It assumes that the first argument has already been parsed and is given by the parameter e.

```
⟨ Declare a function called match_font 64 ⟩
static void mpx_define_font(MPX mpx, web_integer e)
{   /* e is an external font number */
  unsigned i;   /* index into font_num and internal_num */
  web_integer n;   /* length of the font name and area */
  web_integer k;   /* general purpose loop counter */
  web_integer x;   /* a temporary value for scaled size computation */
  if (mpx→nfonts ≡ max_fonts)
    mpx_abort(mpx, "DVItoMP␣capacity␣exceeded␣(max␣fonts=%d)!", max_fonts);
  ⟨ Allocate an index i into the font_num and internal_num tables 60 ⟩;
  ⟨ Read the font parameters into position for font nf 61 ⟩;
  mpx→internal_num[i] = mpx_match_font(mpx, mpx→nfonts, true);
  if (mpx→internal_num[i] ≡ (int) mpx→nfonts) {
    mpx→info_base[mpx→nfonts] = max_widths;   /* indicate that the info isn't loaded yet */
    mpx→local_only[mpx→nfonts] = mpx→vf_reading;
    incr(mpx→nfonts);
  }
}
```

60. ⟨ Allocate an index i into the *font_num* and *internal_num* tables 60 ⟩ ≡

```
if (mpx→vf_ptr ≡ mpx→nfonts)
  mpx_abort(mpx, "DVItoMP␣capacity␣exceeded␣(max␣font␣numbers=%d)", max_fnums);
if (mpx→vf_reading) {
  mpx→font_num[mpx→nfonts] = 0;
  i = mpx→vf_ptr;
  decr(mpx→vf_ptr);
}
else {
  i = mpx→nfonts;
}
mpx→font_num[i] = e
```

This code is used in section 59.

61. ⟨ Read the font parameters into position for font nf 61 ⟩ ≡

```
mpx→font_check_sum[mpx→nfonts] = mpx_signed_quad(mpx);
⟨ Read font_scaled_size[nf] and font_design_size[nf] 62 ⟩;
n = mpx_get_byte(mpx);   /* that is the area */
n = n + mpx_get_byte(mpx);
mpx→font_name[mpx→nfonts] = xmalloc((size_t)(n + 1), 1);
for (k = 0; k < n; k++) mpx→font_name[mpx→nfonts][k] = (char) mpx_get_byte(mpx);
mpx→font_name[mpx→nfonts][k] = 0
```

This code is used in section 59.

62. The scaled size and design size are stored in `DVI` units divided by 2^{20}. The units for scaled size are a little different if we are reading a virtual font, but this will be corrected when the scaled size is used. The scaled size also needs to be truncated to at most 23 significant bits in order to make the character width calculation match what TEX does.

```
⟨ Read font_scaled_size[nf] and font_design_size[nf] 62 ⟩ ≡
  x = mpx_signed_quad(mpx);
  k = 1;
  while (mpx→x > °40000000) {
    x = x/2;
    k = k + k;
  }
  mpx→font_scaled_size[mpx→nfonts] = x * k/1048576.0;
  if (mpx→vf_reading)
    mpx→font_design_size[mpx→nfonts] = mpx_signed_quad(mpx) * mpx→dvi_per_fix/1048576.0;
  else mpx→font_design_size[mpx→nfonts] = mpx_signed_quad(mpx)/1048576.0;
```

This code is used in section 61.

63.
```
⟨ Globals 9 ⟩ +≡
  double dvi_per_fix;      /* converts points scaled 2^20 to DVI units */
```

64. The *match_font* function tries to find a match for the font with internal number *ff*, returning *nf* or the number of the matching font. If *exact* = *true*, the name and scaled size should match. Otherwise the scaled size need not match but the font found must be already loaded, not just defined.

```
⟨ Declare a function called match_font 64 ⟩ ≡
  static web_integer mpx_match_font(MPX mpx, unsigned ff, web_boolean exact)
  {
    unsigned f;      /* font number being tested */
    for (f = 0; f < mpx→nfonts; f++) {
      if (f ≠ ff) {
        ⟨ Compare the names of fonts f and ff; continue if they differ 65 ⟩;
        if (exact) {
          if (fabs(mpx→font_scaled_size[f] − mpx→font_scaled_size[ff]) ≤ font_tolerance) {
            if (¬mpx→vf_reading) {
              if (mpx→local_only[f]) {
                mpx→font_num[f] = mpx→font_num[ff];
                mpx→local_only[f] = false;
              }
              else if (mpx→font_num[f] ≠ mpx→font_num[ff]) {
                continue;
              }
            }
            break;
          }
        }
        else if (mpx→info_base[f] ≠ max_widths) {
          break;
        }
      }
    }
    if (f < mpx→nfonts) {
      ⟨ Make sure fonts f and ff have matching design sizes and checksums 66 ⟩;
    }
    return (web_integer) f;
  }
```

This code is used in section 59.

65. ⟨ Compare the names of fonts *f* and *ff*; **continue** if they differ 65 ⟩ ≡

```
  if (strcmp(mpx→font_name[f], mpx→font_name[ff])) continue
```

This code is used in section 64.

66. ⟨ Make sure fonts *f* and *ff* have matching design sizes and checksums 66 ⟩ ≡

```
  if (fabs(mpx→font_design_size[f] − mpx→font_design_size[ff]) > font_tolerance) {
    font_error("Inconsistent␣design␣sizes␣given␣for␣", ff);
  }
  else if (mpx→font_check_sum[f] ≠ mpx→font_check_sum[ff]) {
    font_warn("Checksum␣mismatch␣for␣", ff);
  }
```

This code is used in section 64.

67. Reading ordinary fonts. An auxiliary array *in_width* is used to hold the widths as they are input. The global variable *tfm_check_sum* is set to the check sum that appears in the current `TFM` file.

```
⟨ Globals 9 ⟩ +≡
  web_integer in_width[256];      /* TFM width data in DVI units */
  web_integer tfm_check_sum;      /* check sum found in tfm_file */
```

68. Here is a procedure that absorbs the necessary information from a `TFM` file, assuming that the file has just been successfully reset so that we are ready to read its first byte. (A complete description of `TFM` file format appears in the documentation of `TFtoPL` and will not be repeated here.) The procedure does not check the `TFM` file for validity, nor does it give explicit information about what is wrong with a `TFM` file that proves to be invalid. The procedure simply aborts the program if it detects anything amiss in the `TFM` data.

```
static void mpx_in_TFM(MPX mpx, web_integer f)
{    /* input TFM data for font f or abort */
  web_integer k;    /* index for loops */
  int lh;    /* length of the header data, in four-byte words */
  int nw;    /* number of words in the width table */
  unsigned int wp;    /* new value of info_ptr after successful input */
  ⟨ Read past the header data; abort if there is a problem 69 ⟩;
  ⟨ Store character-width indices at the end of the width table 70 ⟩;
  ⟨ Read the width values into the in_width table 71 ⟩;
  ⟨ Move the widths from in_width to width 74 ⟩;
  mpx→fbase[f] = 0;
  mpx→ftop[f] = 0;
  mpx→info_ptr = wp;
  mpx_fclose(mpx, mpx→tfm_file);
  return;
}
```

69. ⟨ Read past the header data; *abort* if there is a problem 69 ⟩ ≡

```
mpx_read_tfm_word(mpx);
lh = mpx→b2 * (int)(256) + mpx→b3;
mpx_read_tfm_word(mpx);
mpx→font_bc[f] = mpx→b0 * (int)(256) + mpx→b1;
mpx→font_ec[f] = mpx→b2 * (int)(256) + mpx→b3;
if (mpx→font_ec[f] < mpx→font_bc[f]) mpx→font_bc[f] = mpx→font_ec[f] + 1;
if (mpx→info_ptr + (unsigned int) mpx→font_ec[f] − (unsigned int) mpx→font_bc[f] + 1 > max_widths)
  mpx_abort(mpx, "DVItoMP capacity exceeded (width table size=%d)!", max_widths);
wp = mpx→info_ptr + (unsigned int) mpx→font_ec[f] − (unsigned int) mpx→font_bc[f] + 1;
mpx_read_tfm_word(mpx);
nw = mpx→b0 * 256 + mpx→b1;
if ((nw ≡ 0) ∨ (nw > 256)) font_abort("Bad TFM file for ", f);
for (k = 1; k ≤ 3 + lh; k++) {
  if (feof(mpx→tfm_file)) font_abort("Bad TFM file for ", f);
  mpx_read_tfm_word(mpx);
  if (k ≡ 4) {
    if (mpx→b0 < 128)
      mpx→tfm_check_sum = ((mpx→b0 * (int)(256) + mpx→b1) * 256 + mpx→b2) * 256 + mpx→b3;
    else
      mpx→tfm_check_sum = (((mpx→b0 − 256) * (int)(256) + mpx→b1) * 256 + mpx→b2) * 256 + mpx→b3;
  }
  if (k ≡ 5) {
    if (mpx→mode ≡ mpx_troff_mode) {
      mpx→font_design_size[f] = (((mpx→b0 * (int)(256) + mpx→b1) * 256 + mpx→b2) * 256 +
          mpx→b3)/(65536.0 * 16);
    }
  }
}
```

This code is used in section 68.

70. ⟨ Store character-width indices at the end of the *width* table 70 ⟩ ≡

```
if (wp > 0) {
  for (k = (int) mpx→info_ptr; k ≤ (int) wp − 1; k++) {
    mpx_read_tfm_word(mpx);
    if (mpx→b0 > nw) font_abort("Bad TFM file for ", f);
    mpx→width[k] = mpx→b0;
  }
}
```

This code is used in section 68.

71. No fancy width calculation is needed here because DVItoMP stores widths in their raw form as multiples of the design size scaled by 2^{20}. The *font_scaled_size* entries have been computed so that the final width compution can be done in floating point if enough precision is available.

⟨ Read the width values into the *in_width* table 71 ⟩ ≡

```
for (k = 0; k ≤ nw − 1; k++) {
  mpx_read_tfm_word(mpx);
  if (mpx→b0 > 127) mpx→b0 = mpx→b0 − 256;
  mpx→in_width[k] = ((mpx→b0 * °400 + mpx→b1) * °400 + mpx→b2) * °400 + mpx→b3;
}
```

This code is used in section 68.

72. The width compution uses a scale factor *dvi_scale* that will be introduced later. It is equal to one when not typesetting a character from a virtual font. In that case, the following expressions do the width computation that is so important in `DVItype`. It is less important here because it is impractical to guarantee precise character positioning in MetaPost output. Nevertheless, the width compution will be precise if reals have at least 46-bit mantissas and $round(x - .5)$ is equivalent to $\lfloor x \rfloor$. It may be a good idea to modify this computation if these conditions are not met.

⟨ Width of character c in font f 72 ⟩ ≡
```
floor(mpx→dvi_scale * mpx→font_scaled_size[f] * char_width(f, c))
```
This code is used in section 94.

73. ⟨ Width of character p in font *cur_font* 73 ⟩ ≡
```
floor(mpx→dvi_scale * mpx→font_scaled_size[cur_font] * char_width(cur_font, p))
```
This code is used in section 118.

74. ⟨ Move the widths from *in_width* to *width* 74 ⟩ ≡
```
if (mpx→in_width[0] ≠ 0) font_abort("Bad TFM file for ", f);      /* the first width should be zero */
mpx→info_base[f] = (int)(mpx→info_ptr − (unsigned int) mpx→font_bc[f]);
if (wp > 0) {
  for (k = (int) mpx→info_ptr; k ≤ (int) wp − 1; k++) {
    mpx→width[k] = mpx→in_width[mpx→width[k]];
  }
}
```
This code is used in section 68.

75. Reading virtual fonts.

The *in_VF* procedure absorbs the necessary information from a `VF` file that has just been reset so that we are ready to read the first byte. (A complete description of `VF` file format appears in the documention of `VFtoVP`). Like *in_TFM*, this procedure simply aborts the program if it detects anything wrong with the `VF` file.

```
⟨ Declare a function called first_par 115 ⟩
static void mpx_in_VF(MPX mpx, web_integer f)
{  /* read VF data for font f or abort */
  web_integer p;  /* a byte from the VF file */
  boolean was_vf_reading;  /* old value of vf_reading */
  web_integer c;  /* the current character code */
  web_integer limit;  /* space limitations force character codes to be less than this */
  web_integer w;  /* a TFM width being read */
  was_vf_reading = mpx→vf_reading;
  mpx→vf_reading = true;
  ⟨ Start reading the preamble from a VF file 76 ⟩;
  ⟨ Initialize the data structures for the virtual font 77 ⟩;
  p = mpx_get_byte(mpx);
  while (p ≥ fnt_def1) {
    if (p > fnt_def1 + 3) font_abort("Bad␣VF␣file␣for␣", f);
    mpx_define_font(mpx, mpx_first_par(mpx, (unsigned int) p));
    p = mpx_get_byte(mpx);
  }
  while (p ≤ 242) {
    if (feof(mpx→vf_file)) font_abort("Bad␣VF␣file␣for␣", f);
    ⟨ Read the packet length, character code, and TFM width 78 ⟩;
    ⟨ Store the character packet in cmd_buf 79 ⟩;
    p = mpx_get_byte(mpx);
  }
  if (p ≡ post) {
    ⟨ Finish setting up the data structures for the new virtual font 80 ⟩;
    mpx→vf_reading = was_vf_reading;
    return;
  }
}
```

76. ⟨ Start reading the preamble from a `VF` file 76 ⟩ ≡

```
p = mpx_get_byte(mpx);
if (p ≠ pre) font_abort("Bad␣VF␣file␣for␣", f);
p = mpx_get_byte(mpx);  /* fetch the identification byte */
if (p ≠ 202) font_abort("Bad␣VF␣file␣for␣", f);
p = mpx_get_byte(mpx);  /* fetch the length of the introductory comment */
while (p−− > 0) (void) mpx_get_byte(mpx);
mpx→tfm_check_sum = mpx_signed_quad(mpx);
(void) mpx_signed_quad(mpx);  /* skip over the design size */
```

This code is used in section 75.

77. ⟨ Initialize the data structures for the virtual font 77 ⟩ ≡
```
mpx→ftop[f] = (web_integer) mpx→vf_ptr;
if (mpx→vf_ptr ≡ mpx→nfonts)
  mpx_abort(mpx, "DVItoMP␣capacity␣exceeded␣(max␣font␣numbers=%d)", max_fnums);
decr(mpx→vf_ptr);
mpx→info_base[f] = (web_integer) mpx→info_ptr;
limit = max_widths − mpx→info_base[f];
mpx→font_bc[f] = limit; mpx→font_ec[f] = 0
```
This code is used in section 75.

78. ⟨ Read the packet length, character code, and TFM width 78 ⟩ ≡
```
if (p ≡ 242) {
  p = mpx_signed_quad(mpx);
  c = mpx_signed_quad(mpx);
  w = mpx_signed_quad(mpx);
  if (c < 0) font_abort("Bad␣VF␣file␣for␣", f);
}
else {
  c = mpx_get_byte(mpx);
  w = mpx_get_three_bytes(mpx);
}
if (c ≥ limit) mpx_abort(mpx, "DVItoMP␣capacity␣exceeded␣(max␣widths=%d)", max_widths);
if (c < mpx→font_bc[f]) mpx→font_bc[f] = c;
if (c > mpx→font_ec[f]) mpx→font_ec[f] = c;
char_width(f, c) = w
```
This code is used in section 75.

79. ⟨ Store the character packet in *cmd_buf* 79 ⟩ ≡
```
if (mpx→n_cmds + (unsigned int) p ≥ virtual_space)
  mpx_abort(mpx, "DVItoMP␣capacity␣exceeded␣(virtual␣font␣space=%d)", virtual_space);
start_cmd(f, c) = (web_integer) mpx→n_cmds;
while (p > 0) {
  mpx→cmd_buf[mpx→n_cmds] = (unsigned char) mpx_get_byte(mpx);
  incr(mpx→n_cmds);
  decr(p);
}
mpx→cmd_buf[mpx→n_cmds] = eop;      /* add the end-of-packet marker */
incr(mpx→n_cmds)
```
This code is used in section 75.

80. There are unused *width* and *cmd_ptr* entries if $font\_bc[f] > 0$ but it isn't worthwhile to slide everything down just to save a little space.

⟨ Finish setting up the data structures for the new virtual font 80 ⟩ ≡
```
mpx→fbase[f] = (web_integer)(mpx→vf_ptr + 1); mpx→info_ptr = (unsigned
    int)(mpx→info_base[f] + mpx→font_ec[f] + 1)
```
This code is used in section 75.

81. Loading fonts.
The character width information for a font is loaded when the font is selected for the first time. This information might already be loaded if the font has already been used at a different scale factor. Otherwise, we look for a `VF` file, or failing that, a `TFM` file. All this is done by the *select_font* function that takes an external font number e and returns the corresponding internal font number with the width information loaded.

```
static web_integer mpx_select_font(MPX mpx, web_integer e)
{
  int f;      /* the internal font number */
  int ff;     /* internal font number for an existing version */
  web_integer k;      /* general purpose loop counter */
  ⟨ Set f to the internal font number that corresponds to e, or abort if there is none 82 ⟩;
  if (mpx→info_base[f] ≡ max_widths) {
    ff = mpx_match_font(mpx, (unsigned) f, false);
    if (ff < (int) mpx→nfonts) {
      ⟨ Make font f refer to the width information from font ff 83 ⟩;
    }
    else {
      ⟨ Move the VF file name into the cur_name string 84 ⟩;
      if (mpx_open_vf_file(mpx)) {
        mpx_in_VF(mpx, f);
      }
      else {
        if (¬mpx_open_tfm_file(mpx)) font_abort("No␣TFM␣file␣found␣for␣", f);
        mpx_in_TFM(mpx, f);
      }
      ⟨ Make sure the checksum in the font file matches the one given in the font_def for font f 85 ⟩;
    }
    ⟨ Do any other initialization required for the new font f 99 ⟩;
  }
  return f;
}
```

82. ⟨ Set f to the internal font number that corresponds to e, or *abort* if there is none 82 ⟩ ≡

```
if (mpx→cur_ftop ≤ mpx→nfonts) mpx→cur_ftop = mpx→nfonts;
mpx→font_num[mpx→cur_ftop] = e;
k = (web_integer) mpx→cur_fbase;
while ((mpx→font_num[k] ≠ e) ∨ mpx→local_only[k]) incr(k);
if (k ≡ (int) mpx→cur_ftop) mpx_abort(mpx, "Undefined␣font␣selected");
f = mpx→internal_num[k]
```

This code is used in section 81.

83. ⟨ Make font f refer to the width information from font ff 83 ⟩ ≡

```
{
  mpx→font_bc[f] = mpx→font_bc[ff];
  mpx→font_ec[f] = mpx→font_ec[ff];
  mpx→info_base[f] = mpx→info_base[ff];
  mpx→fbase[f] = mpx→fbase[ff];
  mpx→ftop[f] = mpx→ftop[ff];
}
```

This code is used in section 81.

84. The string *cur_name* is supposed to be set to the external name of the `VF` file for the current font.

```
⟨ Move the VF file name into the cur_name string 84 ⟩ ≡
  mpx→cur_name = xstrdup(mpx→font_name[f])
```

This code is used in section 81.

85.

```
⟨ Make sure the checksum in the font file matches the one given in the font_def for font f 85 ⟩ ≡
  {
    if ((mpx→font_check_sum[f] ≠ 0) ∧ (mpx→tfm_check_sum ≠ 0) ∧
          (mpx→font_check_sum[f] ≠ mpx→tfm_check_sum)) {
      font_warn("Checksum␣mismatch␣for␣", f);
    }
  }
```

This code is used in section 81.

86. Low level output routines.

One of the basic output operations is to write a MetaPost string expression for a sequence of characters to be typeset. The main difficulties are that such strings can contain arbitrary eight-bit bytes and there is no fixed limit on the length of the string that needs to be produced. In extreme cases this can lead to expressions such as

```
char7&char15&char31&"?FWayzz"
&"zzaF"&char15&char3&char31
&"Nxzzzzzzzwvtsqo"
```

87. A global variable *state* keeps track of the output process. When *state* = *normal* we have begun a quoted string and the next character should be a printable character or a closing quote. When *state* = *special* the last thing printed was a "`char`" construction or a closing quote and an ampersand should come next. The starting condition *state* = *initial* is a lot like *state* = *special*, except no ampersand is required.

```
#define special 0 /* the state after printing a "char" expression */
#define normal 1 /* the state value in a quoted string */
#define initial 2 /* initial state */
⟨ Globals 9 ⟩ +≡
  int state; /* controls the process of printing a string */
  int print_col; /* there are at most this many characters on the current line */
```

88.
```
⟨ Set initial values 10 ⟩ +≡
  mpx→state = initial;
  mpx→print_col = 0; /* there are at most this many characters on the current line */
```

89. To print a string on the `MPX` file, initialize *print_col*, ensure that *state* = *initial*, and pass the characters one-at-a-time to *print_char*.

```
⟨ Declare subroutines for printing strings 89 ⟩ ≡
  static void mpx_print_char(MPX mpx, unsigned char c)
  {
    web_integer l; /* number of characters to print c or the char expression */
    if (printable(c)) l = 1;
    else if (c < 10) l = 5;
    else if (c < 100) l = 6;
    else l = 7;
    if (mpx→print_col + l > line_length − 2) {
      if (mpx→state ≡ normal) {
        fprintf(mpx→mpxfile, "\"");
        mpx→state = special;
      }
      fprintf(mpx→mpxfile, "\n");
      mpx→print_col = 0;
    }
    ⟨ Print c and update state and print_col 90 ⟩;
  }
```

See also section 91.

This code is used in section 57.

90. ⟨ Print c and update *state* and *print_col* 90 ⟩ ≡

```
if (mpx→state ≡ normal) {
  if (printable(c)) {
    fprintf(mpx→mpxfile, "%c", xchr(c));
  }
  else {
    fprintf(mpx→mpxfile, "\"&char%d", c);
    mpx→print_col += 2;
  }
}
else {
  if (mpx→state ≡ special) {
    fprintf(mpx→mpxfile, "&");
    incr(mpx→print_col);
  }
  if (printable(c)) {
    fprintf(mpx→mpxfile, "\"%c", xchr(c));
    incr(mpx→print_col);
  }
  else {
    fprintf(mpx→mpxfile, "char%d", c);
  }
}
mpx→print_col += l;
if (printable(c)) mpx→state = normal;
else mpx→state = special
```

This code is used in section 89.

91. The *end_char_string* procedure gets the string ended properly and ensures that there is room for l more characters on the output line.

⟨ Declare subroutines for printing strings 89 ⟩ +≡

```
static void mpx_end_char_string(MPX mpx, web_integer l)
{
  while (mpx→state > special) {
    fprintf(mpx→mpxfile, "\"");
    incr(mpx→print_col);
    decr(mpx→state);
  }
  if (mpx→print_col + l > line_length) {
    fprintf(mpx→mpxfile, "\n␣");
    mpx→print_col = 0;
  }
  mpx→state = initial;      /* get ready to print the next string */
}
```

92. Since *end_char_string* resets *state*: = *initial*, all we have to do is set *state*: = *initial* once at the beginning.

⟨ Set initial values 10 ⟩ +≡

```
mpx→state = initial;
```

93. Characters and rules are positioned according to global variables h and v as will be explained later. We also need scale factors that convert quantities to the right units when they are printed in the `MPX` file.

Even though all variable names in the MetaPost output are made local via `save` commands, it is still desirable to preceed them with underscores. This makes the output more likely to work when used in a macro definition, since the generated variables names must not collide with formal parameters in such cases.

```
⟨ Globals 9 ⟩ +≡
  web_integer h;
  web_integer v;      /* the current position in DVI units */
  double conv;        /* converts DVI units to MetaPost points */
  double mag;         /* magnification factor times 1000 */
```

94. ⟨ Declare a procedure called *finish_last_char* 103 ⟩

```
  static void mpx_do_set_char(MPX mpx, web_integer f, web_integer c)
  {
    if ((c < mpx→font_bc[f]) ∨ (c > mpx→font_ec[f]))
      mpx_abort(mpx, "attempt to typeset invalid character %d", c);
    if ((mpx→h ≠ mpx→str_h2)∨(mpx→v ≠ mpx→str_v)∨(f ≠ mpx→str_f)∨(mpx→dvi_scale ≠ mpx→str_scale))
      {
      if (mpx→str_f ≥ 0) {
        mpx_finish_last_char(mpx);
      }
      else if (¬mpx→fonts_used) {
        ⟨ Prepare to output the first character on a page 98 ⟩;
      }
      if (¬mpx→font_used[f]) ⟨ Prepare to use font f for the first time on a page 102 ⟩;
      fprintf(mpx→mpxfile, "_s(");
      mpx→print_col = 3;
      mpx→str_scale = mpx→dvi_scale;
      mpx→str_f = f;
      mpx→str_v = mpx→v;
      mpx→str_h1 = mpx→h;
    }
    mpx_print_char(mpx, (unsigned char) c);
    mpx→str_h2 = (web_integer)(mpx→h + ⟨ Width of character c in font f 72 ⟩);
  }
```

95. ⟨ Globals 9 ⟩ +≡

```
  boolean font_used[(max_fonts + 1)];   /* has this font been used on this page? */
  boolean fonts_used;     /* has any font been used on this page? */
  boolean rules_used;     /* has any rules been set on this page? */
  web_integer str_h1;
  web_integer str_v;      /* starting position for current output string */
  web_integer str_h2;     /* where the current output string ends */
  web_integer str_f;      /* internal font number for the current output string */
  double str_scale;       /* value of dvi_scale for the current output string */
```

96. Before using any fonts we need to define a MetaPost macro for typesetting character strings. The *font_used* array is not initialized until it is actually time to output a character.

```
⟨ Declarations 20 ⟩ +≡
  static void mpx_prepare_font_use(MPX mpx);
```

97. **static void** *mpx_prepare_font_use*(**MPX** *mpx*)

```
{
  unsigned k;
  for (k = 0; k < mpx→nfonts; k++) mpx→font_used[k] = false;
  mpx→fonts_used = true;
  fprintf(mpx→mpxfile, "string␣_n[];\n");
  fprintf(mpx→mpxfile, "vardef␣_s(expr␣_t,_f,_m,_x,_y)(text␣_c)=\n");
  fprintf(mpx→mpxfile,
      "␣␣addto␣_p␣also␣_t␣infont␣_f␣scaled␣_m␣shifted␣(_x,_y)␣_c;␣enddef;\n");
}
```

98. ⟨ Prepare to output the first character on a page 98 ⟩ ≡

```
mpx_prepare_font_use(mpx)
```

This code is used in section 94.

99. ⟨ Do any other initialization required for the new font *f* 99 ⟩ ≡

```
mpx→font_used[f] = false;
```

This code is used in sections 81 and 192.

100. Do what is necessary when the font with internal number f is used for the first time on a page.

⟨ Declarations 20 ⟩ +≡

```
static void mpx_first_use(MPX mpx, int f);
```

101. **static void** *mpx_first_use*(**MPX** *mpx*, **int** *f*)

```
{
  mpx→font_used[f] = true;
  fprintf(mpx→mpxfile, "_n%d=", f);
  mpx→print_col = 6;
  mpx_print_font(mpx, f);
  mpx_end_char_string(mpx, 1);
  fprintf(mpx→mpxfile, ";\n");
}
```

102. ⟨ Prepare to use font *f* for the first time on a page 102 ⟩ ≡

```
mpx_first_use(mpx, f);
```

This code is used in section 94.

103. We maintain the invariant that $str\_f = -1$ when there is no output string under construction.

```
⟨ Declare a procedure called finish_last_char 103 ⟩ ≡
  static void mpx_finish_last_char(MPX mpx)
  {
    double m, x, y;      /* font scale factor and MetaPost coordinates of reference point */
    if (mpx→str_f ≥ 0) {
      if (mpx→mode ≡ mpx_tex_mode) {
        m = mpx→str_scale * mpx→font_scaled_size[mpx→str_f] * mpx→mag / mpx→font_design_size[mpx→str_f];
        x = mpx→conv * mpx→str_h1;
        y = mpx→conv * (−mpx→str_v);
        if ((fabs(x) ≥ 4096.0) ∨ (fabs(y) ≥ 4096.0) ∨ (m ≥ 4096.0) ∨ (m < 0)) {
          mpx_warn(mpx, "text␣is␣out␣of␣range");
          mpx_end_char_string(mpx, 60);
        }
        else {
          mpx_end_char_string(mpx, 40);
        }
        fprintf(mpx→mpxfile, ",_n%d,%1.5f,%1.4f,%1.4f,", mpx→str_f, m, x, y);
        ⟨ Print a withcolor specifier if appropriate 154 ⟩
        fprintf(mpx→mpxfile, ");\n");
      }
      else {
        m = mpx→str_size / mpx→font_design_size[mpx→str_f];
        x = mpx→dmp_str_h1 * mpx→unit;
        y = YCORR − mpx→dmp_str_v * mpx→unit;
        if (fabs(x) ≥ 4096.0 ∨ fabs(y) ≥ 4096.0 ∨ m ≥ 4096.0 ∨ m < 0) {
          mpx_warn(mpx, "text␣out␣of␣range␣ignored");
          mpx_end_char_string(mpx, 67);
        }
        else {
          mpx_end_char_string(mpx, 47);
        }
        fprintf(mpx→mpxfile, "),␣_n%d", mpx→str_f);
        fprintf(mpx→mpxfile, ",%.5f,%.4f,%.4f)", (m * 1.00375), (x/100.0), y);
        mpx_slant_and_ht(mpx);
        fprintf(mpx→mpxfile, ";\n");
      }
      mpx→str_f = −1;
    }
  }
```

This code is used in section 94.

104. Setting rules is fairly simple.

```
static void mpx_do_set_rule(MPX mpx, web_integer ht, web_integer wd)
{
  double xx1, yy1, xx2, yy2, ww;
    /* MetaPost coordinates of lower-left and upper-right corners */
  if (wd ≡ 1) {⟨ Handle a special rule that determines the box size 106 ⟩}
  else if ((ht > 0) ∨ (wd > 0)) {
    if (mpx→str_f ≥ 0) mpx_finish_last_char(mpx);
    if (¬mpx→rules_used) {
      mpx→rules_used = true;
      fprintf(mpx→mpxfile, "interim␣linecap:=0;\n""vardef␣_r(expr␣_a,_\
          w)(text␣_t)␣=\n""␣␣addto␣_p␣doublepath␣_a␣withpen␣pencircle␣scaled␣_w␣_t␣end\
          def;");
    }
    ⟨ Make (xx1, yy1) and (xx2, yy2) then ends of the desired penstroke and ww the desired stroke
        width 105 ⟩;
    if ((fabs(xx1) ≥ 4096.0) ∨ (fabs(yy1) ≥ 4096.0) ∨
        (fabs(xx2) ≥ 4096.0) ∨ (fabs(yy2) ≥ 4096.0) ∨ (ww ≥ 4096.0))
      mpx_warn(mpx, "hrule␣or␣vrule␣is␣out␣of␣range");
    fprintf(mpx→mpxfile, "_r((%1.4f,%1.4f)..(%1.4f,%1.4f),␣%1.4f,", xx1, yy1, xx2, yy2, ww);
    ⟨ Print a withcolor specifier if appropriate 154 ⟩
    fprintf(mpx→mpxfile, ");\n");
  }
}
```

105. ⟨ Make (xx1, yy1) and (xx2, yy2) then ends of the desired penstroke and ww the desired stroke width 105 ⟩ ≡

```
xx1 = mpx→conv * mpx→h;
yy1 = mpx→conv * (−mpx→v);
if (wd > ht) {
  xx2 = xx1 + mpx→conv * wd;
  ww = mpx→conv * ht;
  yy1 = yy1 + 0.5 * ww;
  yy2 = yy1;
}
else {
  yy2 = yy1 + mpx→conv * ht;
  ww = mpx→conv * wd;
  xx1 = xx1 + 0.5 * ww;
  xx2 = xx1;
}
```

This code is used in section 104.

106. Rules of width one dvi unit are not typeset since `MPtoTeX` adds an extraneous rule of this width in order to allow `DVItoMP` to deduce the dimensions of the boxes it ships out. The box width is the left edge of the last such rule; the height and depth are at the top and bottom of the rule. There should be only one special rule per picture but there could be more if the user tries to typeset his own one-dvi-unit rules. In this case the dimension-determining rule is the last one in the picture.

```
⟨ Handle a special rule that determines the box size 106 ⟩ ≡
  {
    mpx→pic_wd = mpx→h;
    mpx→pic_dp = mpx→v;
    mpx→pic_ht = ht − mpx→v;
  }
```

This code is used in section 104.

107.
```
⟨ Globals 9 ⟩ +≡
  web_integer pic_dp;
  web_integer pic_ht;
  web_integer pic_wd;      /* picture dimensions from special rule */
```

108. The following initialization and clean-up is required. We do a little more initialization than is absolutely necessary since some compilers might complain if the variables are uninitialized when *do_set_char* tests them.

```
  static void mpx_start_picture(MPX mpx)
  {
    mpx→fonts_used = false;
    mpx→rules_used = false;
    mpx→graphics_used = false;
    mpx→str_f = −1;
    mpx→str_v = 0;
    mpx→str_h2 = 0;
    mpx→str_scale = 1.0;      /* values don't matter */
    mpx→dmp_str_v = 0.0;
    mpx→dmp_str_h2 = 0.0;
    mpx→str_size = 0.0;
    fprintf(mpx→mpxfile, "begingroup␣save␣%s_p,_r,_s,_n;␣picture␣_p;␣_p=nullpicture;\n",
        (mpx→mode ≡ mpx_tex_mode ? "" : "_C,_D,"));
  }
  static void mpx_stop_picture(MPX mpx)
  {
    double w, h, dd;      /* width, height, negative depth in PostScript points */
    if (mpx→str_f ≥ 0) mpx_finish_last_char(mpx);
    if (mpx→mode ≡ mpx_tex_mode) {
      ⟨ Print a setbounds command based on picture dimensions 109 ⟩;
    }
    fprintf(mpx→mpxfile, "_p␣endgroup\n");
  }
```

109. ⟨ Print a **setbounds** command based on picture dimensions 109 ⟩ ≡

```
dd = -mpx→pic_dp * mpx→conv;
w = mpx→conv * mpx→pic_wd;
h = mpx→conv * mpx→pic_ht; fprintf(mpx→mpxfile,
    "setbounds␣currentpicture␣to␣(0,%1.4f)--(%1.4f,%1.4f)--\n""␣(%1.4f,%1.4f)--(0,%1.4f)--cycle;\n",
    dd, w, dd, w, h, h)
```

This code is used in section 108.

110. Translation to symbolic form.
The main work of DVItoMP is accomplished by the *do_dvi_commands* procedure, which produces the output for an entire page, assuming that the *bop* command for that page has already been processed. This procedure is essentially an interpretive routine that reads and acts on the DVI commands. It is also capable of executing the typesetting commands for a character in a virtual font.

111. The definition of DVI files refers to six registers, (h, v, w, x, y, z), which hold **web_integer** values in DVI units. These units come directly from the input file except they need to be rescaled when typesetting characters from a virtual font. The stack of (h, v, w, x, y, z) values is represented by six arrays called *hstack*, ..., *zstack*.

```
⟨ Globals 9 ⟩ +≡
  web_integer w;
  web_integer x;
  web_integer y;
  web_integer z;      /* current state values (h and v have already been declared) */
  web_integer hstack[(stack_size + 1)];
  web_integer vstack[(stack_size + 1)];
  web_integer wstack[(stack_size + 1)];
  web_integer xstack[(stack_size + 1)];
  web_integer ystack[(stack_size + 1)];
  web_integer zstack[(stack_size + 1)];      /* pushed down values in DVI units */
  web_integer stk_siz;      /* the current stack size */
  double dvi_scale;      /* converts units of current input source to DVI units */
```

112.
```
⟨ Do initialization required before starting a new page 112 ⟩ ≡
  mpx→dvi_scale = 1.0;
  mpx→stk_siz = 0;
  mpx→h = 0;
  mpx→v = 0;
  mpx→Xslant = 0.0; mpx→Xheight = 0.0
```
This code is used in sections 123 and 207.

113. Next, we need procedures to handle *push* and *pop* commands.

```
⟨ Declare procedures to handle color commands 137 ⟩static void mpx_do_push(MPX mpx)
  {
    if (mpx→stk_siz ≡ stack_size)
      mpx_abort(mpx, "DVItoMP␣capacity␣exceeded␣(stack␣size=%d)", stack_size);
    mpx→hstack[mpx→stk_siz] = mpx→h;
    mpx→vstack[mpx→stk_siz] = mpx→v;
    mpx→wstack[mpx→stk_siz] = mpx→w;
    mpx→xstack[mpx→stk_siz] = mpx→x;
    mpx→ystack[mpx→stk_siz] = mpx→y;
    mpx→zstack[mpx→stk_siz] = mpx→z;
    incr(mpx→stk_siz);
  }
  static void mpx_do_pop(MPX mpx)
  {
    if (mpx→stk_siz ≡ 0) bad_dvi("attempt␣to␣pop␣empty␣stack");
    else {
      decr(mpx→stk_siz);
      mpx→h = mpx→hstack[mpx→stk_siz];
      mpx→v = mpx→vstack[mpx→stk_siz];
      mpx→w = mpx→wstack[mpx→stk_siz];
      mpx→x = mpx→xstack[mpx→stk_siz];
      mpx→y = mpx→ystack[mpx→stk_siz];
      mpx→z = mpx→zstack[mpx→stk_siz];
    }
  }
```

114. The *set_virtual_char* procedure is mutually recursive with *do_dvi_commands*. This is really a supervisory procedure that calls *do_set_char* or adjusts the input source to read typesetting commands for a character in a virtual font.

```
static void mpx_do_dvi_commands(MPX mpx);
static void mpx_set_virtual_char(MPX mpx, web_integer f, web_integer c)
{
  double old_scale;      /* original value of dvi_scale */
  unsigned old_buf_ptr;      /* original value of the input pointer buf_ptr */
  unsigned old_fbase, old_ftop;      /* originally applicable part of the font_num table */
  if (mpx→fbase[f] ≡ 0) mpx_do_set_char(mpx, f, c);
  else {
    old_fbase = mpx→cur_fbase;
    old_ftop = mpx→cur_ftop;
    mpx→cur_fbase = (unsigned int) mpx→fbase[f];
    mpx→cur_ftop = (unsigned int) mpx→ftop[f];
    old_scale = mpx→dvi_scale;
    mpx→dvi_scale = mpx→dvi_scale * mpx→font_scaled_size[f];
    old_buf_ptr = mpx→buf_ptr;
    mpx→buf_ptr = (unsigned int) start_cmd(f, c);
    mpx_do_push(mpx);
    mpx_do_dvi_commands(mpx);
    mpx_do_pop(mpx);
    mpx→buf_ptr = old_buf_ptr;
    mpx→dvi_scale = old_scale;
    mpx→cur_fbase = old_fbase;
    mpx→cur_ftop = old_ftop;
  }
}
```

115. Before we get into the details of *do_dvi_commands*, it is convenient to consider a simpler routine that computes the first parameter of each opcode.

```
#define four_cases(A) (A): case (A)+1: case (A)+2: case (A)+3
#define eight_cases(A) four_cases((A)): case four_cases((A)+4)
#define sixteen_cases(A) eight_cases((A)): case eight_cases((A)+8)
#define thirty_two_cases(A) sixteen_cases((A)): case sixteen_cases((A)+16)
#define sixty_four_cases(A) thirty_two_cases((A)): case thirty_two_cases((A)+32)
```

⟨ Declare a function called *first_par* 115 ⟩ ≡

```
static web_integer mpx_first_par(MPX mpx, unsigned int o)
{
  switch (o) {
  case sixty_four_cases(set_char_0): case sixty_four_cases(set_char_0+64):
    return (web_integer)(o - set_char_0);
    break;
  case set1: case put1: case fnt1: case xxx1: case fnt_def1: return mpx_get_byte(mpx);
    break;
  case set1+1: case put1+1: case fnt1+1: case xxx1+1: case fnt_def1+1:
    return mpx_get_two_bytes(mpx);
    break;
  case set1+2: case put1+2: case fnt1+2: case xxx1+2: case fnt_def1+2:
    return mpx_get_three_bytes(mpx);
    break;
  case right1: case w1: case x1: case down1: case y1: case z1: return mpx_signed_byte(mpx);
    break;
  case right1+1: case w1+1: case x1+1: case down1+1: case y1+1: case z1+1:
    return mpx_signed_pair(mpx);
    break;
  case right1+2: case w1+2: case x1+2: case down1+2: case y1+2: case z1+2:
    return mpx_signed_trio(mpx);
    break;
  case set1+3: case set_rule: case put1+3: case put_rule: case right1+3: case w1+3:
    case x1+3: case down1+3: case y1+3: case z1+3: case fnt1+3: case xxx1+3:
    case fnt_def1+3: return mpx_signed_quad(mpx);
    break;
  case nop: case bop: case eop: case push: case pop: case pre: case post: case post_post:
    case undefined_commands: return 0;
    break;
  case w0: return mpx→w;
    break;
  case x0: return mpx→x;
    break;
  case y0: return mpx→y;
    break;
  case z0: return mpx→z;
    break;
  case sixty_four_cases(fnt_num_0): return (web_integer)(o - fnt_num_0);
    break;
  }
  return 0;      /* compiler warning */
}
```

This code is used in section 75.

116. Here is the *do_dvi_commands* procedure.

```
static void mpx_do_dvi_commands(MPX mpx)
{
  unsigned int o;  /* operation code of the current command */
  web_integer p, q;  /* parameters of the current command */
  web_integer cur_font;  /* current internal font number */
  if ((mpx→cur_fbase < mpx→cur_ftop) ∧ (mpx→buf_ptr < virtual_space))
    cur_font = mpx_select_font(mpx, mpx→font_num[mpx→cur_ftop − 1]);  /* select first local font */
  else cur_font = max_fnums + 1;  /* current font is undefined */
  mpx→w = 0;
  mpx→x = 0;
  mpx→y = 0;
  mpx→z = 0;  /* initialize the state variables */
  while (true) {
    ⟨ Translate the next command in the DVI file; return if it was eop 118 ⟩;
  }
}
```

117. The multiway switch in *first_par*, above, was organized by the length of each command; the one in *do_dvi_commands* is organized by the semantics.

118. ⟨ Translate the next command in the DVI file; **return** if it was *eop* 118 ⟩ ≡

```
{ o = (unsigned int) mpx_get_byte(mpx);
p = mpx_first_par(mpx, o);
if (feof(mpx→dvi_file)) bad_dvi("the␣DVI␣file␣ended␣prematurely");
if (o < set1 + 4) {  /* set_char_0 through set_char_127, set1 through set4 */
if (cur_font > max_fnums) {
  if (mpx→vf_reading) mpx_abort(mpx, "no␣font␣selected␣for␣character␣%d␣in␣virtual␣font", p);
  else bad_dvi_two("no␣font␣selected␣for␣character␣%d", p);
}
mpx_set_virtual_char(mpx, cur_font, p); mpx→h += ⟨ Width of character p in font cur_font 73 ⟩;
}
else {
  switch (o) {
  case four_cases(put1): mpx_set_virtual_char(mpx, cur_font, p);
    break;
  case set_rule: q = (web_integer) trunc(mpx_signed_quad(mpx) * mpx→dvi_scale);
    mpx_do_set_rule(mpx, (web_integer) trunc(p * mpx→dvi_scale), q);
    mpx→h += q;
    break;
  case put_rule: q = (web_integer) trunc(mpx_signed_quad(mpx) * mpx→dvi_scale);
    mpx_do_set_rule(mpx, (web_integer) trunc(p * mpx→dvi_scale), q);
    break;
    ⟨ Additional cases for translating DVI command o with first parameter p 119 ⟩
  case undefined_commands: bad_dvi_two("undefined␣command␣%d", o);
    break;
  }  /* all cases have been enumerated */
}
}
```

This code is used in section 116.

119. ⟨ Additional cases for translating DVI command o with first parameter p 119 ⟩ ≡

```
case four_cases(xxx1): mpx_do_xxx(mpx, p);
  break;
case pre: case post: case post_post: bad_dvi("preamble or postamble within a page!");
  break;
```

See also sections 120, 121, and 122.

This code is used in section 118.

120. ⟨ Additional cases for translating DVI command o with first parameter p 119 ⟩ +≡

```
case nop: break;
case bop: bad_dvi("bop occurred before eop");
  break;
case eop: return;
  break;
case push: mpx_do_push(mpx);
  break;
case pop: mpx_do_pop(mpx);
  break;
```

121. ⟨ Additional cases for translating DVI command o with first parameter p 119 ⟩ +≡

```
case four_cases(right1): mpx→h += trunc(p * mpx→dvi_scale);
  break;
case w0: case four_cases(w1): mpx→w = (web_integer) trunc(p * mpx→dvi_scale);
  mpx→h += mpx→w;
  break;
case x0: case four_cases(x1): mpx→x = (web_integer) trunc(p * mpx→dvi_scale);
  mpx→h += mpx→x;
  break;
case four_cases(down1): mpx→v += trunc(p * mpx→dvi_scale);
  break;
case y0: case four_cases(y1): mpx→y = (web_integer) trunc(p * mpx→dvi_scale);
  mpx→v += mpx→y;
  break;
case z0: case four_cases(z1): mpx→z = (web_integer) trunc(p * mpx→dvi_scale);
  mpx→v += mpx→z;
  break;
```

122. ⟨ Additional cases for translating DVI command o with first parameter p 119 ⟩ +≡

```
case sixty_four_cases(fnt_num_0): case four_cases(fnt1): cur_font = mpx_select_font(mpx, p);
  break;
case four_cases(fnt_def1): mpx_define_font(mpx, p);
  break;
```

123. The main program. Now we are ready to put it all together. This is where `DVItoMP` starts, and where it ends.

```
static int mpx_dvitomp(MPX mpx, char *dviname)
{
  int k;
  mpx→dviname = dviname;
  mpx_open_dvi_file(mpx);
  ⟨ Process the preamble 125 ⟩;
  mpx_open_mpxfile(mpx);
  if (mpx→banner ≠ Λ) fprintf(mpx→mpxfile, "%s\n", mpx→banner);
  while (true) {
    ⟨ Advance to the next bop command 127 ⟩;
    for (k = 0; k ≤ 10; k++) (void) mpx_signed_quad(mpx);
    ⟨ Do initialization required before starting a new page 112 ⟩;
    mpx_start_picture(mpx);
    mpx_do_dvi_commands(mpx);
    if (mpx→stk_siz ≠ 0) bad_dvi("stack␣not␣empty␣at␣end␣of␣page");
    mpx_stop_picture(mpx);
    fprintf(mpx→mpxfile, "mpxbreak\n");
  }
  if (mpx→dvi_file) mpx_fclose(mpx, mpx→dvi_file);
  if (mpx→history ≤ mpx_cksum_trouble) return 0;
  else return mpx→history;
}
```

124. The main program needs a few global variables in order to do its work.

```
⟨ Globals 9 ⟩ +≡
  web_integer k;
  web_integer p;      /* general purpose registers */
  web_integer numerator;
  web_integer denominator;      /* stated conversion ratio */
```

125. ⟨ Process the preamble 125 ⟩ ≡

```
{
  int p;
  p = mpx_get_byte(mpx);      /* fetch the first byte */
  if (p ≠ pre) bad_dvi("First␣byte␣isn""t␣start␣of␣preamble!");
  p = mpx_get_byte(mpx);      /* fetch the identification byte */
  if (p ≠ id_byte) mpx_warn(mpx, "identification␣in␣byte␣1␣should␣be␣%d!", id_byte);
  ⟨ Compute the conversion factor 126 ⟩;
  p = mpx_get_byte(mpx);      /* fetch the length of the introductory comment */
  while (p > 0) {
    decr(p);
    (void) mpx_get_byte(mpx);
  }
}
```

This code is used in section 123.

126. The conversion factor *conv* is figured as follows: There are exactly n/d decimicrons per `DVI` unit, and 254000 decimicrons per inch, and *resolution* pixels per inch. Then we have to adjust this by the stated amount of magnification. No such adjustment is needed for *dvi_per_fix* since it is used to convert design sizes.

```
⟨ Compute the conversion factor 126 ⟩ ≡
  mpx→numerator = mpx_signed_quad(mpx);
  mpx→denominator = mpx_signed_quad(mpx);
  if ((mpx→numerator ≤ 0) ∨ (mpx→denominator ≤ 0)) bad_dvi("bad␣scale␣ratio␣in␣preamble");
  mpx→mag = mpx_signed_quad(mpx)/1000.0;
  if (mpx→mag ≤ 0.0) bad_dvi("magnification␣isn't␣positive");
  mpx→conv = (mpx→numerator/254000.0) * (72.0/mpx→denominator) * mpx→mag;
  mpx→dvi_per_fix = (254000.0/mpx→numerator) * (mpx→denominator/72.27)/1048576.0;
```

This code is used in section 125.

127.

```
⟨ Advance to the next bop command 127 ⟩ ≡
  do {
    int p;
    k = mpx_get_byte(mpx);
    if ((k ≥ fnt_def1) ∧ (k < fnt_def1 + 4)) {
      p = mpx_first_par(mpx, (unsigned int) k);
      mpx_define_font(mpx, p);
      k = nop;
    }
  } while (k ≡ nop);
  if (k ≡ post) break;
  if (k ≠ bop) bad_dvi("missing␣bop");
```

This code is used in section 123.

128. Global filenames.

```
⟨ Globals 9 ⟩ +≡
  char *dviname;
```

129. Color support. These changes support `dvips`-style "`color push NAME`" and "`color pop`" specials. We store a list of named colors, sorted by name, and decorate the relevant drawing commands with "`withcolor (r,g,b)`" specifiers while a color is defined.

130. A constant bounding the size of the named-color array.

```
#define max_named_colors 100 /* maximum number of distinct named colors */
```

131. Then we declare a record for color types.

```
⟨ Types in the outer block 8 ⟩ +≡
  typedef struct named_color_record {
    const char *name; /* color name */
    const char *value; /* text to pass to MetaPost */
  } named_color_record;
```

132. Declare the named-color array itself.

```
⟨ Globals 9 ⟩ +≡
  named_color_record named_colors[(max_named_colors + 1)];
    /* stores information about named colors, in sorted order by name */
  web_integer num_named_colors; /* number of elements of named_colors that are valid */
```

133. This function, used only during initialization, defines a named color.

```
  static void mpx_def_named_color(MPX mpx, const char *n, const char *v)
  {
    mpx→num_named_colors++;
    assert(mpx→num_named_colors < max_named_colors);
    mpx→named_colors[mpx→num_named_colors].name = n;
    mpx→named_colors[mpx→num_named_colors].value = v;
  }
```

134.

```
⟨ Declarations 20 ⟩ +≡
  static void mpx_def_named_color(MPX mpx, const char *n, const char *v);
```

135. During the initialization phase, we define values for all the named colors defined in `colordvi.tex`. CMYK-to-RGB conversion by GhostScript.

This list has to be sorted alphabetically!

⟨ Set initial values 10 ⟩ +≡

```
mpx→num_named_colors = 0;
mpx_def_named_color(mpx, "Apricot", "(1.0, 0.680006, 0.480006)");
mpx_def_named_color(mpx, "Aquamarine", "(0.180006, 1.0, 0.7)");
mpx_def_named_color(mpx, "Bittersweet", "(0.760012, 0.0100122, 0.0)");
mpx_def_named_color(mpx, "Black", "(0.0, 0.0, 0.0)");
mpx_def_named_color(mpx, "Blue", "(0.0, 0.0, 1.0)");
mpx_def_named_color(mpx, "BlueGreen", "(0.15, 1.0, 0.669994)");
mpx_def_named_color(mpx, "BlueViolet", "(0.1, 0.05, 0.960012)");
mpx_def_named_color(mpx, "BrickRed", "(0.719994, 0.0, 0.0)");
mpx_def_named_color(mpx, "Brown", "(0.4, 0.0, 0.0)");
mpx_def_named_color(mpx, "BurntOrange", "(1.0, 0.489988, 0.0)");
mpx_def_named_color(mpx, "CadetBlue", "(0.380006, 0.430006, 0.769994)");
mpx_def_named_color(mpx, "CarnationPink", "(1.0, 0.369994, 1.0)");
mpx_def_named_color(mpx, "Cerulean", "(0.0600122, 0.889988, 1.0)");
mpx_def_named_color(mpx, "CornflowerBlue", "(0.35, 0.869994, 1.0)");
mpx_def_named_color(mpx, "Cyan", "(0.0, 1.0, 1.0)");
mpx_def_named_color(mpx, "Dandelion", "(1.0, 0.710012, 0.160012)");
mpx_def_named_color(mpx, "DarkOrchid", "(0.6, 0.2, 0.8)");
mpx_def_named_color(mpx, "Emerald", "(0.0, 1.0, 0.5)");
mpx_def_named_color(mpx, "ForestGreen", "(0.0, 0.880006, 0.0)");
mpx_def_named_color(mpx, "Fuchsia", "(0.45, 0.00998169, 0.919994)");
mpx_def_named_color(mpx, "Goldenrod", "(1.0, 0.9, 0.160012)");
mpx_def_named_color(mpx, "Gray", "(0.5, 0.5, 0.5)");
mpx_def_named_color(mpx, "Green", "(0.0, 1.0, 0.0)");
mpx_def_named_color(mpx, "GreenYellow", "(0.85, 1.0, 0.310012)");
mpx_def_named_color(mpx, "JungleGreen", "(0.0100122, 1.0, 0.480006)");
mpx_def_named_color(mpx, "Lavender", "(1.0, 0.519994, 1.0)");
mpx_def_named_color(mpx, "LimeGreen", "(0.5, 1.0, 0.0)");
mpx_def_named_color(mpx, "Magenta", "(1.0, 0.0, 1.0)");
mpx_def_named_color(mpx, "Mahogany", "(0.65, 0.0, 0.0)");
mpx_def_named_color(mpx, "Maroon", "(0.680006, 0.0, 0.0)");
mpx_def_named_color(mpx, "Melon", "(1.0, 0.539988, 0.5)");
mpx_def_named_color(mpx, "MidnightBlue", "(0.0, 0.439988, 0.569994)");
mpx_def_named_color(mpx, "Mulberry", "(0.640018, 0.0800061, 0.980006)");
mpx_def_named_color(mpx, "NavyBlue", "(0.0600122, 0.460012, 1.0)");
mpx_def_named_color(mpx, "OliveGreen", "(0.0, 0.6, 0.0)");
mpx_def_named_color(mpx, "Orange", "(1.0, 0.389988, 0.130006)");
mpx_def_named_color(mpx, "OrangeRed", "(1.0, 0.0, 0.5)");
mpx_def_named_color(mpx, "Orchid", "(0.680006, 0.360012, 1.0)");
mpx_def_named_color(mpx, "Peach", "(1.0, 0.5, 0.3)");
mpx_def_named_color(mpx, "Periwinkle", "(0.430006, 0.45, 1.0)");
mpx_def_named_color(mpx, "PineGreen", "(0.0, 0.75, 0.160012)");
mpx_def_named_color(mpx, "Plum", "(0.5, 0.0, 1.0)");
mpx_def_named_color(mpx, "ProcessBlue", "(0.0399878, 1.0, 1.0)");
mpx_def_named_color(mpx, "Purple", "(0.55, 0.139988, 1.0)");
mpx_def_named_color(mpx, "RawSienna", "(0.55, 0.0, 0.0)");
mpx_def_named_color(mpx, "Red", "(1.0, 0.0, 0.0)");
mpx_def_named_color(mpx, "RedOrange", "(1.0, 0.230006, 0.130006)");
```

```
mpx_def_named_color(mpx,"RedViolet","(0.590018, 0.0, 0.660012)");
mpx_def_named_color(mpx,"Rhodamine","(1.0, 0.180006, 1.0)");
mpx_def_named_color(mpx,"RoyalBlue","(0.0, 0.5, 1.0)");
mpx_def_named_color(mpx,"RoyalPurple","(0.25, 0.1, 1.0)");
mpx_def_named_color(mpx,"RubineRed","(1.0, 0.0, 0.869994)");
mpx_def_named_color(mpx,"Salmon","(1.0, 0.469994, 0.619994)");
mpx_def_named_color(mpx,"SeaGreen","(0.310012, 1.0, 0.5)");
mpx_def_named_color(mpx,"Sepia","(0.3, 0.0, 0.0)");
mpx_def_named_color(mpx,"SkyBlue","(0.380006, 1.0, 0.880006)");
mpx_def_named_color(mpx,"SpringGreen","(0.739988, 1.0, 0.239988)");
mpx_def_named_color(mpx,"Tan","(0.860012, 0.580006, 0.439988)");
mpx_def_named_color(mpx,"TealBlue","(0.119994, 0.980006, 0.640018)");
mpx_def_named_color(mpx,"Thistle","(0.880006, 0.410012, 1.0)");
mpx_def_named_color(mpx,"Turquoise","(0.15, 1.0, 0.8)");
mpx_def_named_color(mpx,"Violet","(0.210012, 0.119994, 1.0)");
mpx_def_named_color(mpx,"VioletRed","(1.0, 0.189988, 1.0)");
mpx_def_named_color(mpx,"White","(1.0, 1.0, 1.0)");
mpx_def_named_color(mpx,"WildStrawberry","(1.0, 0.0399878, 0.610012)");
mpx_def_named_color(mpx,"Yellow","(1.0, 1.0, 0.0)");
mpx_def_named_color(mpx,"YellowGreen","(0.560012, 1.0, 0.260012)");
mpx_def_named_color(mpx,"YellowOrange","(1.0, 0.580006, 0.0)");
```

136. Color commands get a separate warning procedure. *warn* sets *history*: = *mpx_warning_given*, which causes a nonzero exit status; but color errors are trivial and should leave the exit status zero.

```
#define color_warn(A) mpx_warn(mpx,A)
#define color_warn_two(A,B) mpx_warn(mpx,"%s%s",A,B)
```

137. The *do_xxx* procedure handles DVI specials (defined with the *xxx1* ... *xxx4* commands).

```
#define XXX_BUF 256
⟨ Declare procedures to handle color commands 137 ⟩ ≡
  static void mpx_do_xxx(MPX mpx, web_integer p){ unsigned char buf[(XXX_BUF + 1)];
      /* FIXME: Fixed size buffer. */
    web_integer l, r, m, k, len;
    boolean found;
    int bufsiz = XXX_BUF;
    len = 0;
    while ((p > 0) ∧ (len < bufsiz)) {
      buf[len] = (unsigned char) mpx_get_byte(mpx);
      decr(p);
      incr(len);
    }
    ⟨ Check whether buf contains a color command; if not, goto XXXX 138 ⟩
    if (p > 0) {
      color_warn("long␣\"color\"␣special␣ignored");
      goto XXXX;
    }
    if (⟨ buf contains a color pop command 140 ⟩) {⟨ Handle a color pop command 144 ⟩}
    else if (⟨ buf contains a color push command 139 ⟩) {⟨ Handle a color push command 145 ⟩}
    else {
      color_warn("unknown␣\"color\"␣special␣ignored");
      goto XXXX;
    }
  XXXX:
    for (k = 1; k ≤ p; k++) (void) mpx_get_byte(mpx);
    }
```

This code is used in section 113.

138.

```
⟨ Check whether buf contains a color command; if not, goto XXXX 138 ⟩ ≡
  if ((len ≤ 5) ∨ (buf[0] ≠ 'c') ∨ (buf[1] ≠ 'o') ∨ (buf[2] ≠ 'l') ∨ (buf[3] ≠ 'o') ∨ (buf[4] ≠
      'r') ∨ (buf[5] ≠ '␣')) goto XXXX;
```

This code is used in section 137.

139.

```
⟨ buf contains a color push command 139 ⟩ ≡
  (len ≥ 11) ∧ (buf[6] ≡ 'p') ∧ (buf[7] ≡ 'u') ∧ (buf[8] ≡ 's') ∧ (buf[9] ≡ 'h') ∧ (buf[10] ≡ '␣')
```

This code is used in section 137.

140.

```
⟨ buf contains a color pop command 140 ⟩ ≡
  (len ≡ 9) ∧ (buf[6] ≡ 'p') ∧ (buf[7] ≡ 'o') ∧ (buf[8] ≡ 'p')
```

This code is used in section 137.

141. The `color push` and `pop` commands imply a color stack, so we need a global variable to hold that stack.

```
#define max_color_stack_depth 10 /* maximum depth of saved color stack */
```

142. Here's the actual stack variables.

```
⟨ Globals 9 ⟩ +≡
  web_integer color_stack_depth;      /* current depth of saved color stack */
  char *color_stack[(max_color_stack_depth + 1)];      /* saved color stack */
```

143. Initialize the stack to empty.

```
⟨ Set initial values 10 ⟩ +≡
  mpx→color_stack_depth = 0;
```

144. `color pop` just pops the stack.

```
⟨ Handle a color pop command 144 ⟩ ≡
  mpx_finish_last_char(mpx);
  if (mpx→color_stack_depth > 0) {
    free(mpx→color_stack[mpx→color_stack_depth]);
    decr(mpx→color_stack_depth);
  }
  else {
    color_warn("color␣stack␣underflow");
  }
```

This code is used in section 137.

145. `color push` pushes a color onto the stack.

```
⟨ Handle a color push command 145 ⟩ ≡
  mpx_finish_last_char(mpx);
  if (mpx→color_stack_depth ≥ max_color_stack_depth) mpx_abort(mpx, "color␣stack␣overflow");
  incr(mpx→color_stack_depth);      /* I don't know how to do string operations in Pascal. */
    /* Skip over extra spaces after 'color push'. */
  l = 11;
  while ((l < len − 1) ∧ (buf[l] ≡ '␣')) incr(l);
  if (⟨ buf[l] contains an rgb command 146 ⟩) {⟨ Handle a color push rgb command 147 ⟩}
  else if (⟨ buf[l] contains a cmyk command 150 ⟩) {⟨ Handle a color push cmyk command 151 ⟩}
  else if (⟨ buf[l] contains a gray command 148 ⟩) {⟨ Handle a color push gray command 149 ⟩}
  else {⟨ Handle a named color push command 153 ⟩}
```

This code is used in section 137.

146.

```
⟨ buf[l] contains an rgb command 146 ⟩ ≡
  (l + 4 < len) ∧ (buf[l] ≡ 'r') ∧ (buf[l + 1] ≡ 'g') ∧ (buf[l + 2] ≡ 'b') ∧ (buf[l + 3] ≡ '␣')
```

This code is used in section 145.

147.

```
⟨ Handle a color push rgb command 147 ⟩ ≡
  l = l + 4;
  while ((l < len) ∧ (buf[l] ≡ '␣')) incr(l);      /* Remove spaces at end of buf */
  while ((len > l) ∧ (buf[len − 1] ≡ '␣')) decr(len);
  mpx→color_stack[mpx→color_stack_depth] = xmalloc((size_t)(len − l + 3), 1);
  k = 0; ⟨ Copy buf[l] to color_stack[color_stack_depth][k] in tuple form 152 ⟩
```

This code is used in section 145.

148.

```
⟨ buf[l] contains a gray command 148 ⟩ ≡
  (l + 5 < len) ∧ (buf[l] ≡ 'g') ∧ (buf[l + 1] ≡ 'r') ∧ (buf[l + 2] ≡ 'a') ∧ (buf[l + 3] ≡ 'y') ∧ (buf[l + 4] ≡ '␣')
```

This code is used in section 145.

149. ⟨ Handle a color push gray command 149 ⟩ ≡

```
l = l + 5;
while ((l < len) ∧ (buf[l] ≡ '␣')) incr(l);      /* Remove spaces at end of buf */
while ((len > l) ∧ (buf[len − 1] ≡ '␣')) decr(len);
mpx→color_stack[mpx→color_stack_depth] = xmalloc((size_t)(len − l + 9), 1);
strcpy(mpx→color_stack[mpx→color_stack_depth], "white*");
k = 6; ⟨ Copy buf[l] to color_stack[color_stack_depth][k] in tuple form 152 ⟩
```

This code is used in section 145.

150. ⟨ *buf*[*l*] contains a cmyk command 150 ⟩ ≡

```
(l+5 < len)∧(buf[l] ≡ 'c')∧(buf[l+1] ≡ 'm')∧(buf[l+2] ≡ 'y')∧(buf[l+3] ≡ 'k')∧(buf[l+4] ≡ '␣')
```

This code is used in section 145.

151. ⟨ Handle a color push cmyk command 151 ⟩ ≡

```
l = l + 5;
while ((l < len) ∧ (buf[l] ≡ '␣')) incr(l);      /* Remove spaces at end of buf */
while ((len > l) ∧ (buf[len − 1] ≡ '␣')) decr(len);
mpx→color_stack[mpx→color_stack_depth] = xmalloc((size_t)(len − l + 7), 1);
strcpy(mpx→color_stack[mpx→color_stack_depth], "cmyk");
k = 4; ⟨ Copy buf[l] to color_stack[color_stack_depth][k] in tuple form 152 ⟩
```

This code is used in section 145.

152. ⟨ Copy *buf*[*l*] to *color_stack*[*color_stack_depth*][*k*] in tuple form 152 ⟩ ≡

```
mpx→color_stack[mpx→color_stack_depth][k] = '(';
incr(k);
while (l < len) {
  if (buf[l] ≡ '␣') {
    mpx→color_stack[mpx→color_stack_depth][k] = ',';
    while ((l < len) ∧ (buf[l] ≡ '␣')) incr(l);
    incr(k);
  }
  else {
    mpx→color_stack[mpx→color_stack_depth][k] = (char) buf[l];
    incr(l);
    incr(k);
  }
}
mpx→color_stack[mpx→color_stack_depth][k] = ')';
mpx→color_stack[mpx→color_stack_depth][k + 1] = 0;
```

This code is used in sections 147, 149, and 151.

153. Binary-search the *named_colors* array, then push the found color onto the stack.

```
⟨ Handle a named color push command 153 ⟩ ≡
  for (k = l; k ≤ len − 1; k++) {
    buf[k − l] = xchr(buf[k]);
  }
  buf[len − l] = 0;      /* clang: never read: len = len - l; */
  l = 1;
  r = mpx→num_named_colors;
  found = false;
  while ((l ≤ r) ∧ ¬found) {
    m = (l + r)/2;
    k = strcmp((char *)(buf), mpx→named_colors[m].name);
    if (k ≡ 0) {
      mpx→color_stack[mpx→color_stack_depth] = xstrdup(mpx→named_colors[m].value);
      found = true;
    }
    else if (k < 0) {
      r = m − 1;
    }
    else {
      l = m + 1;
    }
  }
  if (¬found) {
    color_warn_two("non-hardcoded␣color␣\"%s\"␣in␣\"color␣push\"␣command", buf);
    mpx→color_stack[mpx→color_stack_depth] = xstrdup((char *)(buf));
  }
```

This code is used in section 145.

154. Last but not least, this code snippet prints a `withcolor` specifier for the top of the color stack, if the stack is nonempty.

```
⟨ Print a withcolor specifier if appropriate 154 ⟩ ≡
  if (mpx→color_stack_depth > 0) {
    fprintf(mpx→mpxfile, "␣withcolor␣%s\n", mpx→color_stack[mpx→color_stack_depth]);
  }
```

This code is used in sections 103 and 104.

155. 4. Dmp.

This program reads device-independent troff output files, and converts them into a symbolic form understood by MetaPost. Some of the code was borrowed from DVItoMP. It understands all the D? graphics functions that dpost does but it ignores 'x X' device control functions such as 'x X SetColor:...', 'x X BeginPath:', and 'x X DrawPath:...'.

The output file is a sequence of MetaPost picture expressions, one for every page in the input file. It makes no difference where the input file comes from, but it is intended to process the result of running eqn and troff on the output of MPtoTR. Such a file contains one page for every btex...etex block in the original input. This program then creates a corresponding sequence of MetaPost picture expressions for use as an auxiliary input file. Since MetaPost expects such files to have the extension .mpx, the output is sometimes called an 'mpx' file.

```
#define SHIFTS 100 /* maximum number of characters with special shifts */
#define MAXCHARS 256 /* character codes fall in the range 0..MAXCHARS-1 */
#define is_specchar(c) (¬mpx¬gflag ∧ (c) ≤ 2) /* does charcode c identify a special char? */
#define LWscale 0.03 /* line width for graphics as a fraction of pointsize */
#define YCORR 12.0 /* V coordinate of reference point in (big) points */
⟨ Globals 9 ⟩ +≡
  int next_specfnt[(max_fnums + 1)]; /* used to link special fonts together */
  int shiftchar[SHIFTS]; /* charcode of character to shift, else -1 */
  float shifth[SHIFTS];
  float shiftv[SHIFTS]; /* shift vals/fontsize (y is upward) */
  int shiftptr; /* number of entries in shift tables */
  int shiftbase[(max_fnums + 1)]; /* initial index into shifth,shiftv,shiftchar */
  int specfnt; /* int. num. of first special font (or FCOUNT) */
  int *specf_tail; /* tail of specfnt list (*specf_tail ≡ FCOUNT) */
  float cursize; /* current type size in (big) points */
  unsigned int curfont; /* internal number for current font */
  float Xslant; /* degrees additional slant for all fonts */
  float Xheight; /* yscale fonts to this height if nonzero */
  float sizescale; /* groff font size scaling factor */
  int gflag; /* non-zero if using groff fonts */
  float unit; /* (big) points per troff unit (0 when unset) */
```

156. ⟨ Set initial values 10 ⟩ +≡

```
  mpx¬shiftptr = 0;
  mpx¬specfnt = (max_fnums + 1);
  mpx¬specf_tail = &(mpx¬specfnt);
  mpx¬unit = 0.0;
  mpx¬lnno = 0; /* this is a reset */
  mpx¬gflag = 0;
  mpx¬h = 0;
  mpx¬v = 0;
```

157. ⟨ Makempx header information 157 ⟩ ≡
```
typedef char *(*mpx_file_finder)(MPX, const char *, const char *, int);
enum mpx_filetype {
  mpx_tfm_format,  /* kpse_tfm_format */
  mpx_vf_format,  /* kpse_vf_format */
  mpx_trfontmap_format,  /* kpse_mpsupport_format */
  mpx_trcharadj_format,  /* kpse_mpsupport_format */
  mpx_desc_format,  /* kpse_troff_font_format */
  mpx_fontdesc_format,  /* kpse_troff_font_format */
  mpx_specchar_format  /* kpse_mpsupport_format */
};
```
See also section 225.

This code is used in section 4.

158. ⟨ Globals 9 ⟩ +≡
```
mpx_file_finder find_file;
```

159. ⟨ Declarations 20 ⟩ +≡
```
static char *mpx_find_file(MPX mpx, const char *nam, const char *mode, int ftype);
```

160.
```
static char *mpx_find_file(MPX mpx, const char *nam, const char *mode, int ftype)
{
  (void) mpx;
  if (mode[0] ≠ 'r' ∨ (¬access(nam, R_OK)) ∨ ftype) {
    return strdup(nam);
  }
  return Λ;
}
```

161. ⟨ Set initial values 10 ⟩ +≡
```
mpx→find_file = mpx_find_file;
```

162. ⟨ Declarations 20 ⟩ +≡
```
static FILE *mpx_fsearch(MPX mpx, const char *nam, int format);
```

163.
```
static FILE *mpx_fsearch(MPX mpx, const char *nam, int format)
{
  FILE *f = Λ;
  char *fname = (mpx→find_file)(mpx, nam, "r", format);
  if (fname) {
    f = fopen(fname, "rb");
    mpx_report(mpx, "%p␣=␣fopen(%s,\"rb\")", f, fname);
  }
  return f;
}
```

164. Hash tables (or rather: AVL lists)

165. ⟨ Types in the outer block 8 ⟩ +≡

```
typedef struct {
  char *name;
  int num;
} avl_entry;
```

166.

```
static int mpx_comp_name(void *p, const void *pa, const void *pb)
{
  (void) p;
  return strcmp(((const avl_entry *) pa)→name, ((const avl_entry *) pb)→name);
}
static void *destroy_avl_entry(void *pa)
{
  avl_entry *p;
  p = (avl_entry *) pa;
  free(p→name);
  free(p);
  return Λ;
}
static void *copy_avl_entry(const void *pa)
{    /* never used */
  const avl_entry *p;
  avl_entry *q;
  p = (const avl_entry *) pa;
  q = malloc(sizeof(avl_entry));
  if (q ≠ Λ) {
    q→name = strdup(p→name);
    q→num = p→num;
  }
  return (void *) q;
}
```

167.

```
static avl_tree mpx_avl_create(MPX mpx)
{
  avl_tree t;
  t = avl_create(mpx_comp_name, copy_avl_entry, destroy_avl_entry, malloc, free, Λ);
  if (t ≡ Λ) mpx_abort(mpx, "Memory␣allocation␣failure");
  return t;
}
```

168. The only two operations on AVL lists are finding already existing items, or interning new items. Finding is handled by explicit *avl_find* calls where needed, but it is wise to have a wrapper around *avl_probe* to check for memory errors.

```
static void mpx_avl_probe(MPX mpx, avl_tree tab, avl_entry *p)
{
  avl_entry *r = (avl_entry *) avl_find(p, tab);
  if (r ≡ Λ) {
    if (avl_ins(p, tab, avl_false) < 0) mpx_abort(mpx, "Memory␣allocation␣failure");
  }
}
```

169. Scanning Numbers

The standard functions atoi(), atof(), and sscanf() provide ways of reading numbers from strings but they give no indication of how much of the string is consumed. These homemade versions don't parse scientific notation.

```
⟨ Globals 9 ⟩ +≡
  char *arg_tail;      /* char after the number just gotten; NULL on failure */
```

170.
```
static int mpx_get_int(MPX mpx, char *s)
{
  register int i, d, neg;
  if (s ≡ Λ) goto BAD;
  for (neg = 0; ; s++) {
    if (*s ≡ '-') neg = ¬neg;
    else if (*s ≠ '␣' ∧ *s ≠ '\t') break;
  }
  if (i = *s − '0', 0 > i ∨ i > 9) goto BAD;
  while (d = *++s − '0', 0 ≤ d ∧ d ≤ 9) i = 10 * i + d;
  mpx→arg_tail = s;
  return neg ? −i : i;
BAD: mpx→arg_tail = Λ;
  return 0;
}
```

171. GROFF font description files use octal character codes *groff_font*(5): The code can be any **web_integer**. If it starts with a 0 it will be interpreted as octal; if it starts with 0x or 0X it will be intepreted as hexadecimal.

```
static int mpx_get_int_map(MPX mpx, char *s)
{
  register int i;
  if (s ≡ Λ) goto BAD;
  i = (int) strtol(s, &(mpx→arg_tail), 0);
  if (s ≡ mpx→arg_tail) goto BAD;
  return i;
BAD: mpx→arg_tail = Λ;
  return 0;
}
```

172. Troff output files contain few if any non-*web_integers*, but this program is prepared to read floats whenever they seem reasonable; i.e., when the number is not being used for character positioning. (For non-PostScript applications h and v are usually in pixels and should be *web_integers*.)

```
static float mpx_get_float(MPX mpx, char *s)
{
  register int d, neg, digits;
  register float x, y;
  digits = 0;
  neg = 0;
  x = 0.0;
  if (s ≠ Λ) {
    for (neg = 0; ; s++) {
      if (*s ≡ '-') neg = ¬neg;
      else if (*s ≠ '␣' ∧ *s ≠ '\t') break;
    }
    x = 0.0;
    while (d = *s - '0', 0 ≤ d ∧ d ≤ 9) {
      x = (float) 10.0 * x + (float) d;
      digits++;
      s++;
    }
    if (*s ≡ '.') {
      y = 1.0;
      while (d = *++s - '0', 0 ≤ d ∧ d ≤ 9) {
        y /= (float) 10.0;
        x += y * (float) d;
        digits++;
      }
    }
  }
  if (digits ≡ 0) {
    mpx→arg_tail = Λ;
    return 0.0;
  }
  mpx→arg_tail = s;
  return neg ? -x : x;
}
```

173. GROFF font description files have metrics field of comma-separated *web_integers*. Traditional troff have a float in this position. The value is not used anyway - thus just skip the value, eat all non-space chars.

```
static float mpx_get_float_map(MPX mpx, char *s)
{
  if (s ≠ Λ) {
    while (isspace((unsigned char) *s)) s++;
    while (¬isspace((unsigned char) *s) ∧ *s) s++;
  }
  mpx→arg_tail = s;
  return 0;
}
```

174. Reading Initialization Files

Read the database file, reserve internal font numbers and set the *font_name* entries. Each line in the database file contains $< troff-name > t, PostScript-name > t < TEX-name >$ or just $< troff-name > t, PostScript-name >$ if the TEX name matches the PostScript name. (—t means one or more tabs.)

```
⟨ Globals 9 ⟩ +≡
  avl_tree trfonts;
```

175.

```
static void mpx_read_fmap(MPX mpx, const char *dbase)
{
  FILE *fin;
  avl_entry *tmp;
  char *nam;      /* a font name being read */
  char *buf;
  mpx→nfonts = 0;
  fin = mpx_fsearch(mpx, dbase, mpx_trfontmap_format);
  if (fin ≡ Λ) mpx_abort(mpx, "Cannot␣find␣%s", dbase);
  mpx→trfonts = mpx_avl_create(mpx);
  while ((buf = mpx_getline(mpx, fin)) ≠ Λ) {
    if (mpx→nfonts ≡ (max_fnums + 1)) mpx_abort(mpx, "Need␣to␣increase␣max_fnums");
    nam = buf;
    while (*buf ∧ *buf ≠ '\t') buf++;
    if (nam ≡ buf) continue;
    tmp = xmalloc(sizeof(avl_entry), 1);
    tmp→name = xmalloc(1, (size_t)(buf − nam) + 1);
    strncpy(tmp→name, nam, (unsigned int)(buf − nam));
    tmp→name[(buf − nam)] = '\0';
    tmp→num = (int) mpx→nfonts++;
    assert(avl_ins(tmp, mpx→trfonts, avl_false) > 0);
    if (*buf) {
      buf++;
      while (*buf ≡ '\t') buf++;
      while (*buf ∧ *buf ≠ '\t') buf++;      /* skip over psname */
      while (*buf ≡ '\t') buf++;
      if (*buf) nam = buf;
      while (*buf) buf++;
    }
    mpx→font_name[tmp→num] = xstrdup(nam);
    mpx→font_num[tmp→num] = −1;      /* indicate font is not mounted */
  }
  mpx_fclose(mpx, fin);
}
```

176. Some characters need their coordinates shifted in order to agree with troff's view of the world. Logically, this information belongs in the font description files but it actually resides in a PostScript prolog that the troff output processor dpost reads. Since that file is in PostScript and subject to change, we read the same information from a small auxiliary file that gives shift amounts relative to the font size with y upward.

GROFF NOTE: The PostScript prologue in GNU groff's font directory does not contain any character shift information, so the following function becomes redundant. Simply keeping an empty "trchars.adj" file around will do fine without requiring any changes to this program.

```
static void mpx_read_char_adj(MPX mpx, const char *adjfile)
{
  FILE *fin;
  char buf[200];
  avl_entry tmp, *p;
  unsigned int i;
  fin = mpx_fsearch(mpx, adjfile, mpx_trcharadj_format);
  if (fin ≡ Λ) mpx_abort(mpx, "Cannot␣find␣%s", adjfile);
  for (i = 0; i < mpx→nfonts; i++) mpx→shiftbase[i] = 0;
  while (fgets(buf, 200, fin) ≠ Λ) {
    if (mpx→shiftptr ≡ SHIFTS − 1) mpx_abort(mpx, "Need␣to␣increase␣SHIFTS");
    if (buf[0] ≠ '␣' ∧ buf[0] ≠ '\t') {
      for (i = 0; buf[i] ≠ '\0'; i++)
        if (buf[i] ≡ '\n') buf[i] = '\0';
      mpx→shiftchar[mpx→shiftptr++] = −1;
      tmp.name = buf;
      p = (avl_entry *) avl_find(&tmp, mpx→trfonts);
      if (p ≡ Λ) mpx_abort(mpx, "%s␣refers␣to␣unknown␣font␣%s", adjfile, buf);
          /* clang: dereference null pointer 'p' */
      assert(p);
      mpx→shiftbase[p→num] = mpx→shiftptr;
    }
    else {
      mpx→shiftchar[mpx→shiftptr] = mpx_get_int(mpx, buf);
      mpx→shifth[mpx→shiftptr] = mpx_get_float(mpx, mpx→arg_tail);
      mpx→shiftv[mpx→shiftptr] = −mpx_get_float(mpx, mpx→arg_tail);
      if (mpx→arg_tail ≡ Λ) mpx_abort(mpx, "Bad␣shift␣entry␣:␣\"%s\"", buf);
      mpx→shiftptr++;
    }
  }
  mpx→shiftchar[mpx→shiftptr++] = −1;
  mpx_fclose(mpx, fin);
}
```

177. Read the DESC file of the troff device to gather information about sizescale and whether running under groff.

Ignore all commands not specially handled. This relieves of collecting commands without arguments here and also makes the program more robust in case of future DESC extensions.

```
static void mpx_read_desc(MPX mpx)
{
  const char *const k1[] = {"res", "hor", "vert", "unitwidth", "paperwidth", "paperlength",
      "biggestfont", "spare2", "encoding", Λ};
  const char *const g1[] = {"family", "paperheight", "postpro", "prepro", "print",
      "image_generator", "broken", Λ};
  char cmd[200];
  FILE *fp;
  int i, n;
  fp = mpx_fsearch(mpx, "DESC", mpx_desc_format);
  if (fp ≡ Λ) mpx_abort(mpx, "Cannot␣find␣DESC");
  while (fscanf(fp, "%199s", cmd) ≠ EOF) {
    if (*cmd ≡ '#') {
      while ((i = getc(fp)) ≠ EOF ∧ i ≠ '\n') ;
      continue;
    }
    if (strcmp(cmd, "fonts") ≡ 0) {
      if (fscanf(fp, "%d", &n) ≠ 1) return;
      for (i = 0; i < n; i++)
        if (fscanf(fp, "%*s") ≡ EOF) return;
    }
    else if (strcmp(cmd, "sizes") ≡ 0) {
      while (fscanf(fp, "%d", &n) ≡ 1 ∧ n ≠ 0) ;
    }
    else if (strcmp(cmd, "styles") ≡ 0 ∨ strcmp(cmd, "papersize") ≡ 0) {
      mpx→gflag++;
      while ((i = getc(fp)) ≠ EOF ∧ i ≠ '\n') ;
    }
    else if (strcmp(cmd, "sizescale") ≡ 0) {
      if (fscanf(fp, "%d", &n) ≡ 1) mpx→sizescale = (float) n;
      mpx→gflag++;
    }
    else if (strcmp(cmd, "charset") ≡ 0) {
      return;
    }
    else {
      for (i = 0; k1[i]; i++)
        if (strcmp(cmd, k1[i]) ≡ 0) {
          if (fscanf(fp, "%*s") ≡ EOF) return;
          break;
        }
      if (k1[i] ≡ 0)
        for (i = 0; g1[i]; i++)
          if (strcmp(cmd, g1[i]) ≡ 0) {
            if (fscanf(fp, "%*s") ≡ EOF) return;
            mpx→gflag = 1;
            break;
          }
```

```
    }
  }
}
```

178. Given one line from the character description file for the font with internal number f, save the appropriate data in the charcodes[f] table. A return value of zero indicates a syntax error.

GROFF: GNU groff uses an extended font description file format documented in *groff_font*(5). In order to allow parsing of groff's font files, this function needs to be rewritten as follows:

1. The 'metrics' field parsed by *mpx_get_float*(*lin*); may include a comma-separated list of up to six decimal *web_integers* rather than just a single floating-point number.
2. The 'charcode' field parsed by *lastcode* = *mpx_get_int*(*arg_tail*); may be given either in decimal, octal, or hexadecimal format.

179. ⟨ Globals 9 ⟩ +≡

```
avl_tree charcodes[(max_fnums + 1)];    /* hash tables for translating char names */
```

180.

```
static int mpx_scan_desc_line(MPX mpx, int f, char *lin)
{
  static int lastcode;
  avl_entry *tmp;
  char *s, *t;
  t = lin;
  while (*lin ≠ ' ' ∧ *lin ≠ '\t' ∧ *lin ≠ '\0') lin++;
  if (lin ≡ t) return 1;
  s = xmalloc((size_t)(lin − t + 1), 1);
  strncpy(s, t, (size_t)(lin − t));
  *(s + (lin − t)) = '\0';
  while (*lin ≡ ' ' ∨ *lin ≡ '\t') lin++;
  if (*lin ≡ '"') {
    if (lastcode < MAXCHARS) {
      tmp = xmalloc(sizeof(avl_entry), 1);
      tmp→name = s;
      tmp→num = lastcode;
      mpx_avl_probe(mpx, mpx→charcodes[f], tmp);
    }
  }
  else {
    (void) mpx_get_float_map(mpx, lin);
    (void) mpx_get_int(mpx, mpx→arg_tail);
    lastcode = mpx_get_int_map(mpx, mpx→arg_tail);
    if (mpx→arg_tail ≡ Λ) return 0;
    if (lastcode < MAXCHARS) {
      tmp = xmalloc(sizeof(avl_entry), 1);
      tmp→name = s;
      tmp→num = lastcode;
      mpx_avl_probe(mpx, mpx→charcodes[f], tmp);
    }
  }
  return 1;
}
```

181. Read the font description file for the font with the given troff name and update the data structures. The result is the internal font number.

```
static int mpx_read_fontdesc(MPX mpx, char *nam)
{    /* troff name */
  char buf[200];
  avl_entry tmp, *p;
  FILE *fin;    /* input file */
  int f;    /* internal font number */
  if (mpx->unit ≡ 0.0) mpx_abort(mpx, "Resolution is not set soon enough");
  tmp.name = nam;
  p = (avl_entry *) avl_find(&tmp, mpx->trfonts);
  if (p ≡ Λ) mpx_abort(mpx, "Font was not in map file");
    /* clang: dereference null pointer 'p' */
  assert(p);
  f = p->num;
  fin = mpx_fsearch(mpx, nam, mpx_fontdesc_format);
  if (fin ≡ Λ) mpx_abort(mpx, "Cannot find %s", nam);
  for ( ; ; ) {
    if (fgets(buf, 200, fin) ≡ Λ)
      mpx_abort(mpx, "Description file for %s ends unexpectedly", nam);
    if (strncmp(buf, "special", 7) ≡ 0) {
      *(mpx->specf_tail) = f;
      mpx->next_specfnt[f] = (max_fnums + 1);
      mpx->specf_tail = &(mpx->next_specfnt[f]);
    }
    else if (strncmp(buf, "charset", 7) ≡ 0) break;
  }
  mpx->charcodes[f] = mpx_avl_create(mpx);
  while (fgets(buf, 200, fin) ≠ Λ)
    if (mpx_scan_desc_line(mpx, f, buf) ≡ 0)
      mpx_abort(mpx, "%s has a bad line in its description file: %s", nam, buf);
  mpx_fclose(mpx, fin);
  return f;
}
```

182. Page and Character Output

```
⟨ Globals 9 ⟩ +≡
  boolean graphics_used;    /* nonzero if any graphics seen on this page */
  float dmp_str_h1;
  float dmp_str_v;    /* corrected start pos for current out string */
  float dmp_str_h2;    /* where the current output string ends */
  float str_size;    /* point size for this text string */
```

183. Print any transformations required by the current Xslant and Xheight settings.

```
⟨ Declarations 20 ⟩ +≡
  static void mpx_slant_and_ht(MPX mpx);
```

184. **static void** *mpx_slant_and_ht*(**MPX** *mpx*)

```
{
  int i = 0;
  fprintf(mpx→mpxfile, "(");
  if (mpx→Xslant ≠ 0.0) {
    fprintf(mpx→mpxfile, "␣slanted%.5f", mpx→Xslant);
    i++;
  }
  if (mpx→Xheight ≠ mpx→cursize ∧ mpx→Xheight ≠ 0.0 ∧ mpx→cursize ≠ 0.0) {
    fprintf(mpx→mpxfile, "␣yscaled%.4f", mpx→Xheight/mpx→cursize);
    i++;
  }
  fprintf(mpx→mpxfile, ")");
}
```

185. Output character number c in the font with internal number f.

```
static void mpx_set_num_char(MPX mpx, int f, int c)
{
  float hh, vv;      /* corrected versions of h, v */
  int i;
  hh = (float) mpx→h;
  vv = (float) mpx→v;
  for (i = mpx→shiftbase[f]; mpx→shiftchar[i] ≥ 0 ∧ i < SHIFTS; i++)
    if (mpx→shiftchar[i] ≡ c) {
      hh += (mpx→cursize/mpx→unit) * mpx→shifth[i];
      vv += (mpx→cursize/mpx→unit) * mpx→shiftv[i];
      break;
    }
  if (hh − mpx→dmp_str_h2 ≥ 1.0 ∨ mpx→dmp_str_h2 − hh ≥ 1.0 ∨ vv − mpx→dmp_str_v ≥
      1.0 ∨ mpx→dmp_str_v − vv ≥ 1.0 ∨ f ≠ mpx→str_f ∨ mpx→cursize ≠ mpx→str_size) {
    if (mpx→str_f ≥ 0) mpx_finish_last_char(mpx);
    else if (¬mpx→fonts_used) mpx_prepare_font_use(mpx);      /* first font usage on this page */
    if (¬mpx→font_used[f]) mpx_first_use(mpx, f);      /* first use of font f on this page */
    fprintf(mpx→mpxfile, "_s((");
    mpx→print_col = 3;
    mpx→str_f = f;
    mpx→dmp_str_v = vv;
    mpx→dmp_str_h1 = hh;
    mpx→str_size = mpx→cursize;
  }
  mpx_print_char(mpx, (unsigned char) c);
  mpx→dmp_str_h2 = hh + (float) char_width(f, c);
}
```

186. Output a string.

```
static void mpx_set_string(MPX mpx, char *cname)
{
  float hh;      /* corrected version of h, current horisontal position */
  if (¬*cname) return;
  hh = (float) mpx¬h;
  mpx_set_num_char(mpx, (int) mpx¬curfont, *cname);
  hh += (float) char_width(mpx¬curfont, (int) *cname);
  while (*++cname) {
    mpx_print_char(mpx, (unsigned char) *cname);
    hh += (float) char_width(mpx¬curfont, (int) *cname);
  }
  mpx¬h = (web_integer) floor(hh + 0.5);
  mpx_finish_last_char(mpx);
}
```

187. Special Characters

Given the troff name of a special character, this routine finds its definition and copies it to the MPX file. It also finds the name of the vardef macro and returns that name. The name should be C.¡something¿.

188. TH: A bit of trickery is added here for case-insensitive file systems. This aliasing allows the CHARLIB directory to exist on DVDs, for example. It is a hack, I know. I've stuck to names on TeXLive.

```
#define test_redo_search do
        {
          if (deff ≡ Λ) deff = mpx_fsearch(mpx, cname, mpx_specchar_format);
        }
        while (0)
static char *mpx_copy_spec_char(MPX mpx, char *cname)
{
  FILE *deff;
  int c;
  char *s, *t;
  char specintro[] = "vardef␣";      /* MetaPost name follows this */
  unsigned k = 0;      /* how much of specintro so far */
  if (strcmp(cname, "ao") ≡ 0) {
    deff = mpx_fsearch(mpx, "ao.x", mpx_specchar_format);
    test_redo_search;
  }
  else if (strcmp(cname, "lh") ≡ 0) {
    deff = mpx_fsearch(mpx, "lh.x", mpx_specchar_format);
    test_redo_search;
  }
  else if (strcmp(cname, "~=") ≡ 0) {
    deff = mpx_fsearch(mpx, "twiddle", mpx_specchar_format);
    test_redo_search;
  }
  else {
    deff = mpx_fsearch(mpx, cname, mpx_specchar_format);
  }
  if (deff ≡ Λ) mpx_abort(mpx, "No␣vardef␣in␣charlib/%s", cname);
  while (k < (unsigned) strlen(specintro)) {
    if ((c = getc(deff)) ≡ EOF) mpx_abort(mpx, "No␣vardef␣in␣charlib/%s", cname);
    putc(c, mpx→mpxfile);
    if (c ≡ specintro[k]) k++;
    else k = 0;
  }
  s = xmalloc(mpx→bufsize, 1);
  t = s;
  while ((c = getc(deff)) ≠ '(') {
    if (c ≡ EOF) mpx_abort(mpx, "vardef␣in␣charlib/%s␣has␣no␣arguments", cname);
    putc(c, mpx→mpxfile);
    *t++ = (char) c;
  }
  putc(c, mpx→mpxfile);
  *t++ = '\0';
  while ((c = getc(deff)) ≠ EOF) ;
  putc(c, mpx→mpxfile);
  return s;
}
```

189. When given a character name instead of a number, we need to check if it is a special character and download the definition if necessary. If the character is not in the current font we have to search the special fonts.

```
⟨ Globals 9 ⟩ +≡
  avl_tree spec_tab;
```

190. The *spec_tab* avl table combines character names with macro names.

```
⟨ Types in the outer block 8 ⟩ +≡
  typedef struct {
    char *name;
    char *mac;
  } spec_entry;
```

```
191. static void mpx_set_char(MPX mpx, char *cname)
{
  int f, c;
  avl_entry tmp, *p;
  spec_entry *sp;
  if (*cname ≡ '␣' ∨ *cname ≡ '\t') return;
  f = (int) mpx→curfont;
  tmp.name = cname;
  p = avl_find(&tmp, mpx→charcodes[f]);
  if (p ≡ Λ) {
    for (f = mpx→specfnt; f ≠ (max_fnums + 1); f = mpx→next_specfnt[f]) {
      p = avl_find(&tmp, mpx→charcodes[f]);
      if (p ≠ Λ) goto OUT_LABEL;
    }
    mpx_abort(mpx, "There␣is␣no␣character␣%s", cname);
  }
OUT_LABEL:    /* clang: dereference null pointer 'p' */
  assert(p);
  c = p→num;
  if (¬is_specchar(c)) {
    mpx_set_num_char(mpx, f, c);
  }
  else {
    if (mpx→str_f ≥ 0) mpx_finish_last_char(mpx);
    if (¬mpx→fonts_used) mpx_prepare_font_use(mpx);
    if (¬mpx→font_used[f]) mpx_first_use(mpx, f);
    if (mpx→spec_tab) mpx→spec_tab = mpx_avl_create(mpx);
    sp = xmalloc(sizeof(spec_entry), 1);
    sp→name = cname;
    sp→mac = Λ;
    {
      spec_entry *r = (spec_entry *) avl_find(sp, mpx→spec_tab);
      if (r ≡ Λ) {
        if (avl_ins(sp, mpx→spec_tab, avl_false) < 0) mpx_abort(mpx, "Memory␣allocation␣failure");
      }
    }
    if (sp→mac ≡ Λ) {
      sp→mac = mpx_copy_spec_char(mpx, cname);    /* this won't be NULL */
    }
    fprintf(mpx→mpxfile, "_s(%s(_n%d)", sp→mac, f);
    fprintf(mpx→mpxfile, ",%.5f,%.4f,%.4f)", (mpx→cursize/mpx→font_design_size[f]) * 1.00375,
        (double)(((float) mpx→h * mpx→unit)/100.0), YCORR − (float) mpx→v * mpx→unit);
    mpx_slant_and_ht(mpx);
    fprintf(mpx→mpxfile, ";\n");
  }
}
```

192. Font Definitions
Mount the font with troff name nam at external font number n and read any necessary font files.

```
static void mpx_do_font_def(MPX mpx, int n, char *nam)
{
  int f;
  unsigned k;
  avl_entry tmp, *p;
  tmp.name = nam;
  p = (avl_entry *) avl_find(&tmp, mpx->trfonts);
  if (p ≡ Λ) mpx_abort(mpx, "Font␣%s␣was␣not␣in␣map␣file", nam);
      /* clang: dereference null pointer 'p' */
  assert(p);
  f = p->num;
  if (mpx->charcodes[f] ≡ Λ) {
    mpx_read_fontdesc(mpx, nam);
    mpx->cur_name = xstrdup(mpx->font_name[f]);
    if (¬mpx_open_tfm_file(mpx)) font_abort("No␣TFM␣file␣found␣for␣", f);
    mpx_in_TFM(mpx, f);
  }
  for (k = 0; k < mpx->nfonts; k++)
    if (mpx->font_num[k] ≡ n) mpx->font_num[k] = -1;
  mpx->font_num[f] = n;
  ⟨ Do any other initialization required for the new font f 99 ⟩;
}
```

193. Time on 'makepath pencircle'
Given the control points of a cubic Bernstein polynomial, evaluate it at t.

```
#define Speed ((float)(PI/4.0))
static float mpx_b_eval(const float *xx, float t)
{
  float zz[4];
  register int i, j;
  for (i = 0; i ≤ 3; i++) zz[i] = xx[i];
  for (i = 3; i > 0; i--)
    for (j = 0; j < i; j++) zz[j] += t * (zz[j + 1] - zz[j]);
  return zz[0];
}
```

194. Find the direction angle at time t on the path 'makepath pencircle'. The tables below give the Bezier control points for MetaPost's cubic approximation to the first octant of a unit circle.

```
static const float xx[] = {1.0, 1.0, (float) 0.8946431597, (float) 0.7071067812};
static const float yy[] = {0.0, (float) 0.2652164899, (float) 0.5195704026, (float) 0.7071067812};
```

195.

```
static float mpx_circangle(float t)
{
  float ti;
  ti = (float) floor(t);
  t -= ti;
  return (float) atan(mpx_b_eval(yy, t)/mpx_b_eval(xx, t)) + ti * Speed;
}
```

196. Find the spline parameter where 'makepath pencircle' comes closest to (cos(a)/2,sin(a)/2).

```
static float mpx_circtime(float a)
{
  int i;
  float t;
  t = a/Speed;
  for (i = 2; --i ≥ 0; ) t += (a − mpx_circangle(t))/Speed;
  return t;
}
```

197. Troff Graphics

```
⟨ Globals 9 ⟩ +≡
  float gx;
  float gy;      /* current point for graphics (init. (h,YCORR/mpx-¿unit-v) */
```

198.

```
static void mpx_prepare_graphics(MPX mpx)
{
  fprintf(mpx→mpxfile, "vardef␣_D(expr␣_d)expr␣_q␣=\n");
  fprintf(mpx→mpxfile, "␣addto␣_p␣doublepath␣_q␣withpen␣pencircle␣scaled␣_d;␣enddef;\n");
  mpx→graphics_used = true;
}
```

199. This function prints the current position (gx,gy). Then if it can read dh dv from string s, it increments (gx,gy) and prints "–". By returning the rest of the string s or NULL if nothing could be read from s, it provides the argument for the next iteration.

```
static char *mpx_do_line(MPX mpx, char *s)
{
  float dh, dv;
  fprintf(mpx→mpxfile, "(%.3f,%.3f)", mpx→gx * mpx→unit, mpx→gy * mpx→unit);
  dh = mpx_get_float(mpx, s);
  dv = mpx_get_float(mpx, mpx→arg_tail);
  if (mpx→arg_tail ≡ Λ) return Λ;
  mpx→gx += dh;
  mpx→gy −= dv;
  fprintf(mpx→mpxfile, "--\n");
  return mpx→arg_tail;
}
```

200. Function *spline_seg*( ) reads two pairs of (dh,dv) increments and prints the corresponding quadratic B-spline segment, leaving the ending point to be printed next time. The return value is the string with the first (dh,dv) pair lopped off. If only one pair of increments is found, we prepare to terminate the iteration by printing last time's ending point and returning NULL.

```
static char *mpx_spline_seg(MPX mpx, char *s)
{
  float dh1, dv1, dh2, dv2;
  dh1 = mpx_get_float(mpx, s);
  dv1 = mpx_get_float(mpx, mpx→arg_tail);
  if (mpx→arg_tail ≡ Λ) mpx_abort(mpx, "Missing␣spline␣increments");
  s = mpx→arg_tail;
  fprintf(mpx→mpxfile, "(%.3f,%.3f)", (mpx→gx + .5 * dh1) * mpx→unit, (mpx→gy − .5 * dv1) * mpx→unit);
  mpx→gx += dh1;
  mpx→gy −= dv1;
  dh2 = mpx_get_float(mpx, s);
  dv2 = mpx_get_float(mpx, mpx→arg_tail);
  if (mpx→arg_tail ≡ Λ) return Λ;
  fprintf(mpx→mpxfile, "..\ncontrols␣(%.3f,%.3f)␣and␣(%.3f,%.3f)..\n",
      (mpx→gx − dh1/6.0) * mpx→unit, (mpx→gy + dv1/6.0) * mpx→unit, (mpx→gx + dh2/6.0) * mpx→unit,
      (mpx→gy − dv2/6.0) * mpx→unit);
  return s;
}
```

201. Draw an ellipse with the given major and minor axes.

```
static void mpx_do_ellipse(MPX mpx, float a, float b)
{
  fprintf(mpx→mpxfile, "makepath(pencircle␣xscaled␣%.3f\n␣yscaled␣%.3f", a * mpx→unit,
      b * mpx→unit);
  fprintf(mpx→mpxfile, "␣shifted␣(%.3f,%.3f));\n", (mpx→gx + .5 * a) * mpx→unit, mpx→gy * mpx→unit);
  mpx→gx += a;
}
```

202. Draw a counter-clockwise arc centered at (cx,cy) with initial and final radii (ax,ay) and (bx,by) respectively.

```
static void mpx_do_arc(MPX mpx, float cx, float cy, float ax, float ay, float bx, float by)
{
  float t1, t2;
  t1 = mpx_circtime((float) atan2(ay, ax));
  t2 = mpx_circtime((float) atan2(by, bx));
  if (t2 < t1) t2 += (float) 8.0;
  fprintf(mpx→mpxfile, "subpath␣(%.5f,%.5f)␣of\n", t1, t2);
  fprintf(mpx→mpxfile, "␣makepath(pencircle␣scaled␣%.3f␣shifted␣(%.3f,%.3f));\n",
      2.0 * sqrt(ax * ax + ay * ay) * mpx→unit, cx * mpx→unit, cy * mpx→unit);
  mpx→gx = cx + bx;
  mpx→gy = cy + by;
}
```

203. String s is everything following the initial 'D' in a troff graphics command.

```
static void mpx_do_graphic(MPX mpx, char *s)
{
  float h1, v1, h2, v2;
  mpx_finish_last_char(mpx);
    /* GROFF uses Fd to set fill color for solid drawing objects to the default, so just ignore that. */
  if (s[0] ≡ 'F' ∧ s[1] ≡ 'd') return;
  mpx→gx = (float) mpx→h;
  mpx→gy = (float) YCORR/mpx→unit − ((float) mpx→v);
  if (¬mpx→graphics_used) mpx_prepare_graphics(mpx);
  fprintf(mpx→mpxfile, "D(%.4f)␣", LWscale * mpx→cursize);
  switch (*s++) {
  case 'c': h1 = mpx_get_float(mpx, s);
    if (mpx→arg_tail ≡ Λ) mpx_abort(mpx, "Bad␣argument␣in␣%s", s − 2);
    mpx_do_ellipse(mpx, h1, h1);
    break;
  case 'e': h1 = mpx_get_float(mpx, s);
    v1 = mpx_get_float(mpx, mpx→arg_tail);
    if (mpx→arg_tail ≡ Λ) mpx_abort(mpx, "Bad␣argument␣in␣%s", s − 2);
    mpx_do_ellipse(mpx, h1, v1);
    break;
  case 'A': fprintf(mpx→mpxfile, "reverse␣");      /* fall through */
  case 'a': h1 = mpx_get_float(mpx, s);
    v1 = mpx_get_float(mpx, mpx→arg_tail);
    h2 = mpx_get_float(mpx, mpx→arg_tail);
    v2 = mpx_get_float(mpx, mpx→arg_tail);
    if (mpx→arg_tail ≡ Λ) mpx_abort(mpx, "Bad␣argument␣in␣%s", s − 2);
    mpx_do_arc(mpx, mpx→gx + h1, mpx→gy − v1, −h1, v1, h2, −v2);
    break;
  case 'l': case 'p':
    while (s ≠ Λ) s = mpx_do_line(mpx, s);
    fprintf(mpx→mpxfile, ";\n");
    break;
  case 'q': do s = mpx_spline_seg(mpx, s); while (s ≠ Λ);
    fprintf(mpx→mpxfile, ";\n");
    break;
  case '~': fprintf(mpx→mpxfile, "(%.3f,%.3f)--", mpx→gx * mpx→unit, mpx→gy * mpx→unit);
    do s = mpx_spline_seg(mpx, s); while (s ≠ Λ);
    fprintf(mpx→mpxfile, "--(%.3f,%.3f);\n", mpx→gx * mpx→unit, mpx→gy * mpx→unit);
    break;
  default: mpx_abort(mpx, "Unknown␣drawing␣function␣%s", s − 2);
  }
  mpx→h = (int) floor(mpx→gx + .5);
  mpx→v = (int) floor(YCORR/mpx→unit + .5 − mpx→gy);
}
```

204. Interpreting Troff Output

```
static void mpx_change_font(MPX mpx, int f)
{
  for (mpx→curfont = 0; mpx→curfont < mpx→nfonts; mpx→curfont++)
    if (mpx→font_num[mpx→curfont] ≡ f) return;
  mpx_abort(mpx, "Bad␣font␣setting");
}
```

205. String s0 is everything following the initial 'x' in a troff device control command. A zero result indicates a stop command.

```
static int mpx_do_x_cmd(MPX mpx, char *s0)
{
  float x;
  int n;
  char *s;
  s = s0;
  while (*s ≡ ' ' ∨ *s ≡ '\t') s++;
  switch (*s++) {
  case 'r':
    if (mpx→unit ≠ 0.0) mpx_abort(mpx, "Attempt to reset resolution");
    while (*s ≠ ' ' ∧ *s ≠ '\t') s++;
    mpx→unit = mpx_get_float(mpx, s);
    if (mpx→unit ≤ 0.0) mpx_abort(mpx, "Bad resolution: x %s", s0);
    mpx→unit = (float) 72.0/mpx→unit;
    break;
  case 'f':
    while (*s ≠ ' ' ∧ *s ≠ '\t') s++;
    n = mpx_get_int(mpx, s);
    if (mpx→arg_tail ≡ Λ) mpx_abort(mpx, "Bad font def: x %s", s0);
    s = mpx→arg_tail;
    while (*s ≡ ' ' ∨ *s ≡ '\t') s++;
    mpx_do_font_def(mpx, n, s);
    break;
  case 's': return 0;
  case 'H':
    while (*s ≠ ' ' ∧ *s ≠ '\t') s++;
    mpx→Xheight = mpx_get_float(mpx, s);
      /* GROFF troff output is scaled groff_out(5): The argument to the s command is in scaled points
         (units of points/n, where n is the argument to the sizescale command in the DESC file.) The
         argument to the x Height command is also in scaled points. sizescale for groff devps is 1000 */
    if (mpx→sizescale ≠ 0.0) {
      if (mpx→unit ≠ 0.0) mpx→Xheight *= mpx→unit;      /* ??? */
      else mpx→Xheight /= mpx→sizescale;
    }
    if (mpx→Xheight ≡ mpx→cursize) mpx→Xheight = 0.0;
    break;
  case 'S':
    while (*s ≠ ' ' ∧ *s ≠ '\t') s++;
    mpx→Xslant = mpx_get_float(mpx, s) * ((float) PI/(float) 180.0);
    x = (float) cos(mpx→Xslant);
    if (−1 · 10^−4 < x ∧ x < 1 · 10^−4) mpx_abort(mpx, "Excessive slant");
    mpx→Xslant = (float) sin(mpx→Xslant)/x;
    break;
  default:      /* do nothing */
    ;
  }
  return 1;
}
```

206. This routine reads commands from the troff output file up to and including the next 'p' or 'x s' command. It also calls *set_num_char*( ) and *set_char*( ) to generate output when appropriate. A zero result indicates that there are no more pages to do.

GROFF: GNU groff uses an extended device-independent output file format documented in *groff_out*(5). In order to allow parsing of groff's output files, this function either needs to be extended to support the new command codes, or else the use of the "t" and "u" commands must be disabled by removing the line "tcommand" from the DESC file in the $(prefix)/lib/groff/devps directory.

```
static int mpx_do_page(MPX mpx, FILE *trf)
{
  char *buf;
  char a, *c, *cc;
  mpx→h = mpx→v = 0;
  while ((buf = mpx_getline(mpx, trf)) ≠ Λ) {
    mpx→lnno++;
    c = buf;
    while (*c ≠ '\0') {
      switch (*c) {
      case '␣': case '\t': case 'w': c++;
        break;
      case 's': mpx→cursize = mpx_get_float(mpx, c + 1);      /* GROFF troff output is scaled
            groff_out(5): The argument to the s command is in scaled points (units of points/n, where
            n is the argument to the sizescale command in the DESC file.) The argument to the x
            Height command is also in scaled points. sizescale for groff devps is 1000 */
        if (mpx→sizescale ≠ 0.0) {
          if (mpx→unit ≠ 0.0) mpx→cursize *= mpx→unit;      /* ??? */
          else mpx→cursize /= mpx→sizescale;
        }
        goto iarg;
      case 'f': mpx_change_font(mpx, mpx_get_int(mpx, c + 1));
        goto iarg;
      case 'c':
        if (c[1] ≡ '\0') mpx_abort(mpx, "Bad␣c␣command␣in␣troff␣output");
        cc = c + 2;
        goto set;
      case 'C': cc = c;
        do cc++; while (*cc ≠ '␣' ∧ *cc ≠ '\t' ∧ *cc ≠ '\0');
        goto set;
      case 'N': mpx_set_num_char(mpx, (int) mpx→curfont, mpx_get_int(mpx, c + 1));
        goto iarg;
      case 'H': mpx→h = mpx_get_int(mpx, c + 1);
        goto iarg;
      case 'V': mpx→v = mpx_get_int(mpx, c + 1);
        goto iarg;
      case 'h': mpx→h += mpx_get_int(mpx, c + 1);
        goto iarg;
      case 'v': mpx→v += mpx_get_int(mpx, c + 1);
        goto iarg;
      case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8':
        case '9':
        if (c[1] < '0' ∨ c[1] > '9' ∨ c[2] ≡ '\0')
          mpx_abort(mpx, "Bad␣nnc␣command␣in␣troff␣output");
        mpx→h += 10 * (c[0] - '0') + c[1] - '0';
```

```
      c++;
      cc = c + 2;
      goto set;
    case 'p': return 1;
    case 'n': (void) mpx_get_int(mpx, c + 1);
      (void) mpx_get_int(mpx, mpx→arg_tail);
      goto iarg;
    case 'D': mpx_do_graphic(mpx, c + 1);
      goto eoln;
    case 'x':
      if (¬mpx_do_x_cmd(mpx, c + 1)) return 0;
      goto eoln;
    case '#': goto eoln;
    case 'F':     /* GROFF uses this command to report filename */
      goto eoln;
    case 'm':     /* GROFF uses this command to control color */
      goto eoln;
    case 'u':     /* GROFF uses this command to output a word with additional white space
          between characters, not implemented */
      mpx_abort(mpx, "Bad␣command␣in␣troff␣output\n""change␣the␣DESC␣fil\
          e␣for␣your␣GROFF␣PostScript␣device,␣remove␣tcommand");
    case 't':     /* GROFF uses this command to output a word */
      cc = c;
      do cc++; while (*cc ≠ '␣' ∧ *cc ≠ '\t' ∧ *cc ≠ '\0');
      a = *cc;
      *cc = '\0';
      mpx_set_string(mpx, ++c);
      c = cc;
      *c = a;
      continue;
    default: mpx_abort(mpx, "Bad␣command␣in␣troff␣output");
    }
    continue;
  set: a = *cc;
    *cc = '\0';
    mpx_set_char(mpx, ++c);
    c = cc;
    *c = a;
    continue;
  iarg: c = mpx→arg_tail;
  }
eoln:     /* do nothing */
  ;
}
return 0;
}
```

207. Main Dmp Program

```
#define dbname "trfonts.map"  /* file for table of troff TFM font names */
#define adjname "trchars.adj"  /* file for character shift amounts */
  static int mpx_dmp(MPX mpx, char *infile)
  {
    int more;
    FILE *trf = mpx_xfopen(mpx, infile, "r");
    mpx_read_desc(mpx);
    mpx_read_fmap(mpx, dbname);
    if (¬mpx→gflag) mpx_read_char_adj(mpx, adjname);
    mpx_open_mpxfile(mpx);
    if (mpx→banner ≠ Λ) fprintf(mpx→mpxfile, "%s\n", mpx→banner);
    if (mpx_do_page(mpx, trf)) {
      do {
        ⟨ Do initialization required before starting a new page 112 ⟩;
        mpx_start_picture(mpx);
        more = mpx_do_page(mpx, trf);
        mpx_stop_picture(mpx);
        fprintf(mpx→mpxfile, "mpxbreak\n");
      } while (more);
    }
    mpx_fclose(mpx, trf);
    if (mpx→history ≤ mpx_cksum_trouble) return 0;
    else return mpx→history;
  }
```

208. 5. Makempx.

Make an MPX file from the labels in a MetaPost source file, using mpto and either dvitomp (TeX) or dmp (troff).

Started from a shell script initially based on John Hobby's original version, that was then translated to C by Akira Kakuto (Aug 1997, Aug 2001), and updated and largely rewritten by Taco Hoekwater (Nov 2006).

Differences between the script and this C version:

The script trapped HUP, INT, QUIT and TERM for cleaning up temporary files. This is a refinement, and not portable.

The script put its own directory in front of the executable search PATH. This is not portable either, and it seems a safe bet that normal users do not have 'mpto', 'dvitomp', or 'dmp' commands in their path.

The command-line '-troff' now also accepts an optional argument.

The troff infile for error diagnostics is renamed "mpxerr.i", not plain "mpxerr".

The original script deleted mpx*.* in the cleanup process.

That is a bit harder in C, because it requires reading the contents of the current directory. The current program assumes that opendir(), readdir() and closedir() are known everywhere where the function getcwd() exists (except on WIN32, where it uses *_findfirst* **c**o).

If this assumption is false, you can define `NO_GETCWD`, and makempx will revert to trying to delete only a few known extensions

There is a -debug switch, preventing the removal of tmp files

```
#define TMPNAME_EXT(a,b)
        {
          strcpy(a,tmpname);
          strcat(a,b);
        }
#define TEXERR "mpxerr.tex"
#define DVIERR "mpxerr.dvi"
#define TROFF_INERR "mpxerr.i"
#define TROFF_OUTERR "mpxerr.t"
```

209. `static void mpx_rename(MPX mpx, const char *a, const char *b)`

```
{
  mpx_report(mpx,"renaming␣%s␣to␣%s",a,b);
  rename(a,b);
}
```

210. ⟨ Globals 9 ⟩ +≡

```
char tex[15];
int debug;
const char *progname;
```

211. Cleaning up

```
static void mpx_default_erasetmp(MPX mpx)
{
  char *wrk;
  char *p;
  if (mpx→mode ≡ mpx_tex_mode) {
    wrk = xstrdup(mpx→tex);
    p = strrchr(wrk, '.');
    *p = '\0';
    strcat(wrk, ".aux");
    remove(wrk);
    *p = '\0';
    strcat(wrk, ".pdf");
    remove(wrk);
    *p = '\0';
    strcat(wrk, ".toc");
    remove(wrk);
    *p = '\0';
    strcat(wrk, ".idx");
    remove(wrk);
    *p = '\0';
    strcat(wrk, ".ent");
    remove(wrk);
    *p = '\0';
    strcat(wrk, ".out");
    remove(wrk);
    *p = '\0';
    strcat(wrk, ".nav");
    remove(wrk);
    *p = '\0';
    strcat(wrk, ".snm");
    remove(wrk);
    *p = '\0';
    strcat(wrk, ".tui");
    remove(wrk);
    free(wrk);
  }
}
```

212. ⟨ Declarations 20 ⟩ +≡

```
static void mpx_erasetmp(MPX mpx);
```

213. **static void** *mpx_cleandir*(**MPX** *mpx*, **char** ∗*cur_path*)

```
{
  char *wrk, *p;
#ifdef _WIN32
  struct _finddata_t c_file;
  long hFile;
#else
  struct dirent *entry;
  DIR * d;
#endif
  wrk = xstrdup(mpx→tex);
  p = strrchr(wrk, '.');
  *p = '\0';      /* now wrk is identical to tmpname */
#ifdef _WIN32
  strcat(cur_path, "/*");
  if ((hFile = _findfirst(cur_path, &c_file)) ≡ -1L) {
    mpx_default_erasetmp(mpx);
  }
  else {
    if (strstr(c_file.name, wrk) ≡ c_file.name) remove(c_file.name);
    while (_findnext(hFile, &c_file) ≠ -1L) {
      if (strstr(c_file.name, wrk) ≡ c_file.name) remove(c_file.name);
    }
    _findclose(hFile);      /* no more entries =¿ close directory */
  }
#else
  if ((d = opendir(cur_path)) ≡ Λ) {
    mpx_default_erasetmp(mpx);
  }
  else {
    while ((entry = readdir(d)) ≠ Λ) {
      if (strstr(entry→d_name, wrk) ≡ entry→d_name) remove(entry→d_name);
    }
    closedir(d);
  }
#endif
  free(wrk);
}
```

214. It is important that *mpx_erasetmp* remains silent. If it find trouble, it should just ignore it.

The string *cur_path* is a little bit larger than needed, because that allows the win32 code in *cleandir* to add the slash and asterisk for globbing without having to reallocate the variable first.

```
#ifdef WIN32
#define GETCWD _getcwd
#else
#define GETCWD getcwd
#endif
  static void mpx_erasetmp(MPX mpx)
  {
    char cur_path[1024];
    if (mpx→debug) return;
    if (mpx→tex[0] ≠ '\0') {
      remove(mpx→tex);
      if (GETCWD(cur_path, 1020) ≡ Λ) {
        mpx_default_erasetmp(mpx);      /* don't know where we are */
      }
      else {
        mpx_cleandir(mpx, cur_path);
      }
    }
  }
```

215. Running the external typesetters.

First, here is a helper for messaging.

```
static char *mpx_print_command(MPX mpx, int cmdlength, char **cmdline)
{
  char *s, *t;
  int i;
  size_t l;
  (void) mpx;
  l = 0;
  for (i = 0; i < cmdlength; i++) {
    l += strlen(cmdline[i]) + 1;
  }
  s = xmalloc((size_t) l, 1);
  t = s;
  for (i = 0; i < cmdlength; i++) {
    if (i > 0) *t++ = ' ';
    t = strcpy(t, cmdline[i]);
    t += strlen(cmdline[i]);
  }
  return s;
}
```

216. This function unifies the external program calling across Posix-like and Win32 systems.

```
static int do_spawn(MPX mpx, char *icmd, char **options)
{
#ifndef WIN32
  pid_t child;
#endif
  int retcode = -1;
  char *cmd = xmalloc(strlen(icmd) + 1, 1);
  if (icmd[0] ≠ '"') {
    strcpy(cmd, icmd);
  }
  else {
    strncpy(cmd, icmd + 1, strlen(icmd) - 2);
    cmd[strlen(icmd) - 2] = 0;
  }
#ifndef WIN32
  child = fork( );
  if (child < 0) mpx_abort(mpx, "fork failed: %s", strerror(errno));
  if (child ≡ 0) {
    if (execvp(cmd, options)) mpx_abort(mpx, "exec failed: %s", strerror(errno));
  }
  else {
    if (wait(&retcode) ≡ child) {
      retcode = (WIFEXITED(retcode) ? WEXITSTATUS(retcode) : -1);
    }
    else {
      mpx_abort(mpx, "wait failed: %s", strerror(errno));
    }
  }
#else
  retcode = _spawnvp(_P_WAIT, cmd, (const char *const *) options);
#endif
  xfree(cmd);
  return retcode;
}
```

217.

```
#ifdef WIN32
#define nuldev "nul"
#else
#define nuldev "/dev/null"
#endif
  static int mpx_run_command(MPX mpx, char *inname, char *outname, int count, char **cmdl)
  {
    char *s;
    int retcode;
    int sav_o, sav_i;      /* for I/O redirection */
    FILE *fr, *fw;      /* read and write streams for the command */
    if (count < 1 ∨ cmdl ≡ Λ ∨ cmdl[0] ≡ Λ) return -1;
        /* return non-zero by default, signalling an error */
    s = mpx_print_command(mpx, count, cmdl);
    mpx_report(mpx, "running␣command␣%s", s);
    free(s);
    fr = mpx_xfopen(mpx, (inname ? inname : nuldev), "r");
    fw = mpx_xfopen(mpx, (outname ? outname : nuldev), "wb");
    ⟨ Save and redirect the standard I/O 219 ⟩;
    retcode = do_spawn(mpx, cmdl[0], cmdl);
    ⟨ Restore the standard I/O 220 ⟩;
    mpx_fclose(mpx, fr);
    mpx_fclose(mpx, fw);
    return retcode;
  }
```

218.

219. Running Troff is more likely than not a series of pipes that feed input to each other. Makempx does all of this itself by using temporary files inbetween. That means we have to juggle about with *stdin* and *stdout*.

This is the only non-ansi C bit of makempx.

⟨ Save and redirect the standard I/O 219 ⟩ ≡

```
#ifdef WIN32
#define DUP _dup
#define DUPP _dup2
#else
#define DUP dup
#define DUPP dup2
#endif
  sav_i = DUP(fileno(stdin));
  sav_o = DUP(fileno(stdout));
  DUPP(fileno(fr), fileno(stdin)); DUPP(fileno(fw), fileno(stdout))
```

This code is used in section 217.

220. ⟨ Restore the standard I/O 220 ⟩ ≡

```
  DUPP(sav_i, fileno(stdin));
  close(sav_i);
  DUPP(sav_o, fileno(stdout)); close(sav_o)
```

This code is used in section 217.

221. The allocation of the array pointed to by *cmdline_addr* is of course much larger than is really needed, but it will still only be a few hunderd bytes at the most, and this ensures that the separate parts of the *maincmd* will all fit.

```
#define split_command(a,b) mpx_do_split_command(mpx,a,&b,' ')
#define split_pipes(a,b) mpx_do_split_command(mpx,a,&b,'|')
static int mpx_do_split_command(MPX mpx,char *maincmd,char ***cmdline_addr,char target)
{
  char *piece;
  char *cmd;
  char **cmdline;
  size_t i;
  int ret = 0;
  int in_string = 0;
  if (strlen(maincmd) ≡ 0) return 0;
  i = sizeof(char *) * (strlen(maincmd) + 1);
  cmdline = xmalloc(i,1);
  memset(cmdline,0,i);
  *cmdline_addr = cmdline;
  i = 0;
  while (maincmd[i] ≡ ' ') i++;
  cmd = xstrdup(maincmd);
  piece = cmd;
  for ( ; i ≤ strlen(maincmd); i++) {
    if (in_string ≡ 1) {
      if (cmd[i] ≡ '"') {
        in_string = 0;
      }
    }
    else if (in_string ≡ 2) {
      if (cmd[i] ≡ '\'') {
        in_string = 0;
      }
    }
    else {
      if (cmd[i] ≡ '"') {
        in_string = 1;
      }
      else if (cmd[i] ≡ '\'') {
        in_string = 2;
      }
      else if (cmd[i] ≡ target) {
        cmd[i] = 0;
        cmdline[ret++] = piece;
        while (i < strlen(maincmd) ∧ cmd[(i + 1)] ≡ ' ') i++;
        piece = cmd + i + 1;
      }
    }
  }
  if (*piece) {
    cmdline[ret++] = piece;
  }
```

```
  return ret;
}
```

222. ⟨ Globals 9 ⟩ +≡

```
char *maincmd;      /* TeX command name */
```

223.

```
static void mpx_command_cleanup(MPX mpx, char **cmdline)
{
  (void) mpx;
  xfree(cmdline[0]);
  xfree(cmdline);
}
```

224.

```
static void mpx_command_error(MPX mpx, int cmdlength, char **cmdline)
{
  char *s = mpx_print_command(mpx, cmdlength, cmdline);
  mpx_command_cleanup(mpx, cmdline);
  mpx_abort(mpx, "Command␣failed:␣%s;␣see␣mpxerr.log", s);
}
```

225. ⟨ Makempx header information 157 ⟩ +≡

```
typedef struct mpx_options {
  int mode;
  char *cmd;
  char *mptexpre;
  char *mpname;
  char *mpxname;
  char *banner;
  int debug;
  mpx_file_finder find_file;
} mpx_options;
int mpx_makempx(mpx_options *mpxopt);
int mpx_run_dvitomp(mpx_options *mpxopt);
```

226.

```
#define ERRLOG "mpxerr.log"
#define MPXLOG "makempx.log"
  int mpx_makempx(mpx_options *mpxopt)
  {
    MPX mpx;
    char **cmdline, **cmdbits;
    char infile[15];
    int retcode, i;
    char tmpname[] = "mpXXXXXX";
    int cmdlength = 1;
    int cmdbitlength = 1;
    if (¬mpxopt→debug) {
      ⟨ Check if mp file is newer than mpxfile, exit if not 229 ⟩;
    }
    mpx = malloc(sizeof(struct mpx_data));
    if (mpx ≡ Λ ∨ mpxopt→cmd ≡ Λ ∨ mpxopt→mpname ≡ Λ ∨ mpxopt→mpxname ≡ Λ)
      return mpx_fatal_error;
    mpx_initialize(mpx);
    if (mpxopt→banner ≠ Λ) mpx→banner = mpxopt→banner;
    mpx→mode = mpxopt→mode;
    mpx→debug = mpxopt→debug;
    if (mpxopt→find_file ≠ Λ) mpx→find_file = mpxopt→find_file;
    if (mpxopt→cmd ≠ Λ) mpx→maincmd = xstrdup(mpxopt→cmd);      /* valgrind says this leaks */
    mpx→mpname = xstrdup(mpxopt→mpname);
    mpx→mpxname = xstrdup(mpxopt→mpxname);
    ⟨ Install and test the non-local jump buffer 18 ⟩;
    if (mpx→debug) {
      mpx→errfile = stderr;
    }
    else {
      mpx→errfile = mpx_xfopen(mpx, MPXLOG, "wb");
    }
    mpx→progname = "makempx";
    ⟨ Initialize the tmpname variable 230 ⟩;
    if (mpxopt→mptexpre ≡ Λ) mpxopt→mptexpre = xstrdup("mptexpre.tex");
    ⟨ Run mpto on the mp file 32 ⟩;
    if (mpxopt→cmd ≡ Λ) goto DONE;
    if (mpx→mode ≡ mpx_tex_mode) {
      ⟨ Run TEX and set up infile or abort 227 ⟩;
      if (mpx_dvitomp(mpx, infile)) {
        mpx_rename(mpx, infile, DVIERR);
        if (¬mpx→debug) remove(mpx→mpxname);
        mpx_abort(mpx, "Dvi␣conversion␣failed:␣%s␣%s\n", DVIERR, mpx→mpxname);
      }
    }
    else if (mpx→mode ≡ mpx_troff_mode) {
      ⟨ Run Troff and set up infile or abort 228 ⟩;
      if (mpx_dmp(mpx, infile)) {
        mpx_rename(mpx, infile, TROFF_OUTERR);
        mpx_rename(mpx, mpx→tex, TROFF_INERR);
```

```
      if (¬mpx→debug) remove(mpx→mpxname);
      mpx_abort(mpx, "Troff␣conversion␣failed:␣%s␣%s\n", TROFF_OUTERR, mpx→mpxname);
    }
  }
  mpx_fclose(mpx, mpx→mpxfile);
  if (¬mpx→debug) mpx_fclose(mpx, mpx→errfile);
  if (¬mpx→debug) {
    remove(MPXLOG);
    remove(ERRLOG);
    remove(infile);
  }
  mpx_erasetmp(mpx);
DONE: retcode = mpx→history;
  mpx_xfree(mpx→buf);
  mpx_xfree(mpx→maincmd);
  for (i = 0; i < (int) mpx→nfonts; i++) mpx_xfree(mpx→font_name[i]);
  free(mpx);
  if (retcode ≡ mpx_cksum_trouble) retcode = 0;
  return retcode;
}
int mpx_run_dvitomp(mpx_options *mpxopt)
{
  MPX mpx;
  int retcode, i;
  mpx = malloc(sizeof(struct mpx_data));
  if (mpx ≡ Λ ∨ mpxopt→mpname ≡ Λ ∨ mpxopt→mpxname ≡ Λ) return mpx_fatal_error;
  mpx_initialize(mpx);
  if (mpxopt→banner ≠ Λ) mpx→banner = mpxopt→banner;
  mpx→mode = mpxopt→mode;
  mpx→debug = mpxopt→debug;
  if (mpxopt→find_file ≠ Λ) mpx→find_file = mpxopt→find_file;
  mpx→mpname = xstrdup(mpxopt→mpname);
  mpx→mpxname = xstrdup(mpxopt→mpxname);
  ⟨Install and test the non-local jump buffer 18⟩;
  if (mpx→debug) {
    mpx→errfile = stderr;
  }
  else {
    mpx→errfile = mpx_xfopen(mpx, MPXLOG, "wb");
  }
  mpx→progname = "dvitomp";
  if (mpx_dvitomp(mpx, mpx→mpname)) {
    if (¬mpx→debug) remove(mpx→mpxname);
    mpx_abort(mpx, "Dvi␣conversion␣failed:␣%s␣%s\n", DVIERR, mpx→mpxname);
  }
  mpx_fclose(mpx, mpx→mpxfile);
  if (¬mpx→debug) mpx_fclose(mpx, mpx→errfile);
  if (¬mpx→debug) {
    remove(MPXLOG);
    remove(ERRLOG);
  }
  mpx_erasetmp(mpx);
```

```
  retcode = mpx→history;
  mpx_xfree(mpx→buf);
  for (i = 0; i < (int) mpx→nfonts; i++) mpx_xfree(mpx→font_name[i]);
  free(mpx);
  if (retcode ≡ mpx_cksum_trouble) retcode = 0;
  return retcode;
}
```

227. TEX has to operate on an actual input file, so we have to append that to the command line.

```
⟨ Run TEX and set up infile or abort 227 ⟩ ≡
{
  char log[15];
  mpx→maincmd = xrealloc(mpx→maincmd, strlen(mpx→maincmd) + strlen(mpx→tex) + 2, 1);
  strcat(mpx→maincmd, "␣");
  strcat(mpx→maincmd, mpx→tex);
  cmdlength = split_command(mpx→maincmd, cmdline);
  retcode = mpx_run_command(mpx, Λ, Λ, cmdlength, cmdline);
  TMPNAME_EXT(log, ".log");
  if (¬retcode) {
    TMPNAME_EXT(infile, ".dvi");
    remove(log);
  }
  else {
    mpx_rename(mpx, mpx→tex, TEXERR);
    mpx_rename(mpx, log, ERRLOG);
    mpx_command_error(mpx, cmdlength, cmdline);
  }
  mpx_command_cleanup(mpx, cmdline);
}
```

This code is used in section 226.

228. ⟨ Run *Troff* and set up *infile* or abort 228 ⟩ ≡

```
{
  char *cur_in, *cur_out;
  char tmp_a[15], tmp_b[15];
  TMPNAME_EXT(tmp_a, ".t");
  TMPNAME_EXT(tmp_b, ".tmp");
  cur_in = mpx→tex;
  cur_out = tmp_a;      /* split the command in bits */
  cmdbitlength = split_pipes(mpx→maincmd, cmdbits);
  cmdline = Λ;
  for (i = 0; i < cmdbitlength; i++) {
    if (cmdline ≠ Λ) free(cmdline);
    cmdlength = split_command(cmdbits[i], cmdline);
    retcode = mpx_run_command(mpx, cur_in, cur_out, cmdlength, cmdline);
    if (retcode) {
      mpx_rename(mpx, mpx→tex, TROFF_INERR);
      mpx_command_error(mpx, cmdlength, cmdline);
    }
    if (i < cmdbitlength − 1) {
      if (i % 2 ≡ 0) {
        cur_in = tmp_a;
        cur_out = tmp_b;
      }
      else {
        cur_in = tmp_b;
        cur_out = tmp_a;
      }
    }
  }
  if (tmp_a ≠ cur_out) {
    remove(tmp_a);
  }
  if (tmp_b ≠ cur_out) {
    remove(tmp_b);
  }
  strcpy(infile, cur_out);
}
```

This code is used in section 226.

229. If MPX file is up-to-date or if MP file does not exist, do nothing.

⟨ Check if mp file is newer than mpxfile, exit if not 229 ⟩ ≡

```
if (mpx_newer(mpxopt→mpname, mpxopt→mpxname)) return 0
```

This code is used in section 226.

230. The splint comment is here because this use of *sprintf* ( ) is definately safe

```
⟨ Initialize the tmpname variable 230 ⟩ ≡
  ␣/*@-bufferoverflowhigh@*/␣
#ifdef HAVE_MKSTEMP
  i = mkstemp(tmpname);
  if (i ≡ −1) {
    sprintf(tmpname, "mp%06d", (int)(time(Λ) % 1000000));
  }
  else {
    close(i);
    remove(tmpname);
  }
#else
#ifdef HAVE_MKTEMP
  {
    char *tmpstring = mktemp(tmpname);
    if ((tmpstring ≡ Λ) ∨ strlen(tmpname) ≡ 0) {
      sprintf(tmpname, "mp%06d", (int)(time(Λ) % 1000000));
    }
    else {      /* this should not really be needed, but better safe than sorry. */
      if (tmpstring ≠ tmpname) {
        i = strlen(tmpstring);
        if (i > 8) i = 8;
        strncpy(tmpname, tmpstring, i);
      }
    }
  }
#else
  sprintf(tmpname, "mp%06d", (int)(time(Λ) % 1000000));
#endif
#endif
  ␣/*@+bufferoverflowhigh@*/␣
```

This code is used in section 226.

⟨ Read three bytes into a, b, and c 52 ⟩ Used in section 49.
⟨ Read two bytes into a and b 51 ⟩ Used in section 49.
⟨ Read *font_scaled_size*[*nf*] and *font_design_size*[*nf*] 62 ⟩ Used in section 61.
⟨ Restore the standard I/O 220 ⟩ Used in section 217.
⟨ Run *TEX* and set up *infile* or abort 227 ⟩ Used in section 226.
⟨ Run *Troff* and set up *infile* or abort 228 ⟩ Used in section 226.
⟨ Run *mpto* on the mp file 32 ⟩ Used in section 226.
⟨ Save and redirect the standard I/O 219 ⟩ Used in section 217.
⟨ Set initial values 10, 24, 48, 56, 88, 92, 135, 143, 156, 161 ⟩ Used in section 7.
⟨ Set f to the internal font number that corresponds to e, or *abort* if there is none 82 ⟩ Used in section 81.
⟨ Start reading the preamble from a `VF` file 76 ⟩ Used in section 75.
⟨ Store character-width indices at the end of the *width* table 70 ⟩ Used in section 68.
⟨ Store the character packet in *cmd_buf* 79 ⟩ Used in section 75.
⟨ Translate the next command in the `DVI` file; **return** if it was *eop* 118 ⟩ Used in section 116.
⟨ Types in the outer block 8, 131, 165, 190 ⟩ Used in section 5.
⟨ Width of character c in font f 72 ⟩ Used in section 94.
⟨ Width of character p in font *cur_font* 73 ⟩ Used in section 118.
⟨ `mpxout.h` 4 ⟩
⟨ *buf*[l] contains a cmyk command 150 ⟩ Used in section 145.
⟨ *buf*[l] contains a gray command 148 ⟩ Used in section 145.
⟨ *buf*[l] contains an rgb command 146 ⟩ Used in section 145.
⟨ *buf* contains a color pop command 140 ⟩ Used in section 137.
⟨ *buf* contains a color push command 139 ⟩ Used in section 137.

Creating mpx files