1. Introduction.

This is the `hbf2gf` program by Werner Lemberg (`wl@gnu.org`).

The "banner line" defined here should be changed whenever `hbf2gf` is modified.

#define *banner* `"hbf2gf␣(CJK␣ver.␣4.3.0)"`

2.

`hbf2gf` is intended to convert Hanzi Bitmap Fonts (HBF) into TEX generic font files (`GF` files) according to the *CJK* package, which `hbf2gf` is part of.

The outline of `hbf2gf` is simple: a CJK (Chinese/Japanese/Korean) bitmap file will be scaled and written in at most *nmb_files* `GF` files, each file containing 256 characters (except the last and possibly the first one). In the normal case it's not necessary to compute the right value of *nmb_files* because `hbf2gf` will do this; you should use −1 instead to indicate this. See the last section for an example.

Alternatively you can call `hbf2gf` similar to METAFONT, i.e., the program will compute one font on demand. This mode will be used if two or three input parameters instead of one are given: the font name, the horizontal resolution, and optionally a vertical scaling factor or resolution to allow modes for e.g. 300×600 dpi printers. `hbf2gf` will extract the configuration file name from the font name; if this file isn't found, the program exits with error code 2 (this is useful for scripts like `mktexpk`). If the configuration file is found but an error occurs while computing the font, error code 1 is returned. In case of success, the exit code is zero.

The characters in the input font files are completely described by the HBF header file. This program uses the HBF API implementation of Ross Paterson (`ross@soi.city.ac.uk`; with small extensions). You will find a description of the HBF standard at `ftp.ifcss.org`.

A batch file created by `hbf2gf` too (if the program computes a whole set of subfonts) will convert the `GF` files to `PK` files using `GFtoPK`, a part of every TEX package.

```
#define TRUE 1
#define FALSE 0

#define STRING_LENGTH 255   /* the maximal length of an input string in the configuration file */
#define FILE_NAME_LENGTH 1024   /* the maximal length (including the path) of a filename */
```

⟨ Global variables 2 ⟩ ≡

```
  int nmb_files = -1;   /* create all files by default */
  int unicode = FALSE;   /* whether a Unicode font should be processed */

  int testing = FALSE;   /* whether we test only the font name */

  int mf_like = FALSE;   /* whether we are in the METAFONT-like mode */
  int file_number = 0;   /* the subfont number */

  double x_resolution = 0.0;   /* the second and third input parameter */
  double y_scale = 1.0;

  int pk_files = TRUE;   /* command line options */
  int tfm_files = TRUE;
  int long_extension = TRUE;
  int quiet = FALSE;

  char config_file[FILE_NAME_LENGTH + 4 + 1];   /* we probably must add '.cfg' */
  char output_name[STRING_LENGTH + 1];

  FILE *config, *out;
  HBF *hbf;
```

```
#ifdef msdos                          /* if we compile under DOS or OS/2 */
#define WRITE_BIN "wb"
#define WRITE_TXT "wt"
#define READ_BIN "rb"
#define READ_TXT "rt"
#else
#define WRITE_BIN "w"
#define WRITE_TXT "w"
#define READ_BIN "r"
#define READ_TXT "r"
#endif
  int end_of_file = FALSE;
```

See also sections 15, 19, 27, 38, 49, 53, 58, 60, 62, 70, 73, and 76.

This code is used in section 4.

3.

One PL file will be created additionally, which describes the font metrics in a readable way. Because all CJK characters have identical bounding boxes, one metrics file is enough—the batch job created by hbf2gf calls PLtoTF to produce this TFM file and then copies it into *nmb_files* metrics files. There usually will be a discrepancy between the number of characters in the last GF file and the TFM file, but this does not harm.

If you specify the ofm_file option in the configuration file, an extended virtual property file (such files have the extension .ovp) for the Ω system is written; this will be then converted with ovp2ovf into an OFM and an OVF file to map all the subfonts into one large virtual font.

4. The main routine.

The main routine takes *file_name*, *x_resolution*, and *y_scale* as command line parameters if in METAFONT-like mode, otherwise *config_file* as the only argument. *read_config*( ) scans the configuration file and fills the global variables, *write_file*( ) writes the GF files, *write_pl*( ) and *write_ovp*( ) write the PL and OVP files respectively, and *write_job*( ) the batch file.

```
⟨ Include files 10 ⟩
⟨ Prototypes 11 ⟩
⟨ Global variables 2 ⟩

int main(argc, argv)
    int argc;                                        /* argument count */
    char *argv[];                                    /* argument values */
  { char *p;

    ⟨ Initialize TeX file searching 78 ⟩

    ⟨ Scan options 7 ⟩

    if (!quiet)
      printf("\n%s\n\n", banner);

    strncpy(config_file, argv[1], FILE_NAME_LENGTH);
    config_file[FILE_NAME_LENGTH] = '\0';

    if (argc > 2 ∨ testing)
     { int l = strlen(config_file);

      if (l > 2)
        config_file[l − 2] = '\0';       /* strip subfont number from file name */
      else
       { if (!quiet)
           printf("`%s'␣can't␣be␣a␣subfont␣created␣by␣hbf2gf\n", config_file);
         exit(2);
       }

      mf_like = TRUE;
     }

    read_config();                        /* will call exit(1) on errors */

    if (mf_like)
      ⟨ Check other arguments 8 ⟩

    ⟨ Initialize variables 28 ⟩

    ⟨ Write files 9 ⟩

    if (tfm_files)
      write_pl();
    if (ofm_file)
      write_ovp();
    if (!mf_like)
      write_job();

    hbfClose(hbf);
```

```
  exit(0);
  return 0;                                                    /* never reached */
}
```

5.

```
#define VERSION
        "\n"
        "Copyright␣(C)␣1996-1999␣Werner␣Lemberg.\n"
        "There␣is␣NO␣warranty.␣␣You␣may␣redistribute␣this␣software\n"
        "under␣the␣terms␣of␣the␣GNU␣General␣Public␣License\n"
        "and␣the␣HBF␣library␣copyright.\n"
        "\n"
        "For␣more␣information␣about␣these␣matters,␣see␣the␣files\n"
        "named␣COPYING␣and␣hbf.c.\n"
        "\n"
```

⟨ Print version 5 ⟩ ≡

```
{printf("\n");
 printf(banner);
 printf("␣(%s)\n", TeX_search_version());
 printf(VERSION);
 exit(0);
}
```

This code is used in section 7.

6.

```
#define USAGE
        "\n"
        "Usage:␣hbf2gf␣[-q]␣configuration_file[.cfg]\n"
        "␣␣␣␣␣␣␣hbf2gf␣[options]␣font_name␣x_resolution␣[y_scale␣|␣y_resolution]\n"
        "␣␣␣␣␣␣␣hbf2gf␣-t␣[-q]␣font_name\n"
        "\n"
        "␣␣Convert␣a␣font␣in␣HBF␣format␣to␣TeX's␣GF␣resp.␣PK␣format.\n"
        "\n"
        "␣␣␣␣␣␣␣␣␣-q␣␣␣␣␣␣␣␣␣␣␣␣␣be␣silent\n"
        "␣␣␣␣␣␣␣␣␣-p␣␣␣␣␣␣␣␣␣␣␣␣␣don't␣produce␣a␣PL␣file\n"
        "␣␣␣␣␣␣␣␣␣-g␣␣␣␣␣␣␣␣␣␣␣␣␣don't␣produce␣a␣GF␣file\n"
        "␣␣␣␣␣␣␣␣␣-n␣␣␣␣␣␣␣␣␣␣␣␣␣use␣no␣resolution␣in␣extension␣(only␣`.gf')\n"
        "␣␣␣␣␣␣␣␣␣-t␣␣␣␣␣␣␣␣␣␣␣␣␣test␣for␣font_name␣(returns␣0␣on␣success)\n"
        "␣␣␣␣␣␣␣␣␣--help␣␣␣␣␣␣␣␣␣print␣this␣message␣and␣exit\n"
        "␣␣␣␣␣␣␣␣␣--version␣␣␣␣␣␣print␣version␣number␣and␣exit\n"
        "\n"
```

⟨ Print help information 6 ⟩ ≡

```
{printf(USAGE);
 exit(0);
}
```

This code is used in section 7.

7.

Three options can be specified to the program (`-p`, `-g`, and `-n`) if in METAFONT-like mode to suppress creation of a `PL` resp. a `GF` file, and to force a '`.gf`' extension (instead of e.g. '`.300gf`'). The corresponding setting of a particular switch in the configuration file is ignored then.

Additionally, the option `-t` tests whether the specified subfont name leads to an **hbf2gf** configuration file. It returns 0 on success and prints out the name of that configuration file (provided the `-q` switch isn't set). This test isn't a thorough one; it only removes the last two characters and checks whether a configuration file with that name exists.

```
⟨ Scan options 7 ⟩ ≡
  if (argc ≡ 2)
   {if (strcmp(argv[1], "--help") ≡ 0)
       ⟨ Print help information 6 ⟩
    else if (strcmp(argv[1], "--version") ≡ 0)
       ⟨ Print version 5 ⟩
   }
  while (argc > 1)
   {p = argv[1];
    if (p[0] ≠ '-')
       break;
    if (p[1] ≡ 'p')
       tfm_files = FALSE;
    else if (p[1] ≡ 'g')
       pk_files = FALSE;
    else if (p[1] ≡ 'n')
       long_extension = FALSE;
    else if (p[1] ≡ 'q')
       quiet = TRUE;
    else if (p[1] ≡ 't')
       testing = TRUE;

    argv++;
    argc--;
   }
  if (testing)
   {if (argc ≠ 2)
      {fprintf(stderr, "Need␣exactly␣one␣parameter␣for␣'-t'␣option.\n");
       fprintf(stderr, "Try␣'hbf2gf␣--help'␣for␣more␣information.\n");
       exit(1);
      }
   }
  else if (argc < 2 ∨ argc > 4)
   {fprintf(stderr, "Invalid␣number␣of␣parameters.\n");
    fprintf(stderr, "Try␣'hbf2gf␣--help'␣for␣more␣information.\n");
    exit(1);
   }
```

This code is used in section 4.

8.

If the (optional) argument is larger than 10, we treat it as a value for the vertical resolution (in dpi), otherwise as a vertical scaling factor.

⟨ Check other arguments 8 ⟩ ≡

```
{if (unicode)
    file_number = (int) strtol(&argv[1][strlen(argv[1]) − 2], (char **) Λ, 16);
  else
    file_number = atoi(&argv[1][strlen(argv[1]) − 2]);

  x_resolution = atof(argv[2]);
  if (x_resolution < PRINTER_MIN_RES_X)
   {fprintf(stderr, "Invalid␣horizontal␣resolution\n");
    exit(1);
   }

  if (argc > 3)
   {y_scale = atof(argv[3]);
    if (y_scale < 0.01)
     {fprintf(stderr, "Invalid␣vertical␣scaling␣factor␣or␣resolution\n");
      exit(1);
     }
    if (y_scale > 10.0)
      y_scale = (double) x_resolution/y_scale;
   }
}
```

This code is used in section 4.

9.

If *unicode* is TRUE, the start value of the running number appended to the base name of the output font files is taken from the HBF header file, otherwise it starts with '01'. *min_char* represents the lower bound of the code range.

If we are in METAFONT-like mode, *file_number* is taken from the command line, and *max_numb* will be set to 1.

⟨ Write files 9 ⟩ ≡

```
{int j, max_numb;

  if (!mf_like)
   {file_number = (unicode ≡ TRUE ? (min_char ≫ 8) : 1);
    if (nmb_files ≡ −1)
      max_numb = (unicode ≡ TRUE ? #100 : 100);
    else
      max_numb = nmb_files;
   }
  else
    max_numb = 1;

  for (j = 0; (j < max_numb) ∧ !end_of_file; file_number++, j++)
    write_file();
```

```
    nmb_files = j;                    /* the real number of output font files */
  }
```

This code is used in section 4.

10.

⟨ Include files 10 ⟩ ≡

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#ifdef TM_IN_SYS_TIME
#include <sys/time.h>
#endif
#include "hbf.h"
```

See also section 69.

This code is used in section 4.

11. The functions.

The first function to be described is *write_file*( ). Each `GF` file consists of three sections: a preamble, a data section, and a postamble. The functions *write_pre*( ), *write_data*( ), and *write_post*( ) handle this.

⟨ Prototypes 11 ⟩ ≡
```
void write_file(void);
```

See also sections 13, 16, 20, 25, 29, 36, 39, 41, 45, 47, 50, 54, 64, 67, 71, 74, and 79.

This code is used in section 4.

12.

In METAFONT-like mode we create font file name extensions similar to METAFONT if the `-n` option isn't specified; otherwise only '`.gf`' will be appended.

```
void write_file(void)
{char output_file[FILE_NAME_LENGTH + 1];

  if (pk_files)
   {if (mf_like)
     {if (unicode)
         sprintf(output_file, "%s%02x.%.0igf",
         output_name, file_number, long_extension ? (int) (x_resolution + 0.5) : 0);
      else
         sprintf(output_file, "%s%02i.%.0igf",
         output_name, file_number, long_extension ? (int) (x_resolution + 0.5) : 0);
     }
    else
     {if (unicode)
         sprintf(output_file, "%s%02x.gf", output_name, file_number);
      else
         sprintf(output_file, "%s%02i.gf", output_name, file_number);
     }
    if (!(out = fopen(output_file, WRITE_BIN)))
     {fprintf(stderr, "Couldn't␣open␣`%s'\n", output_file);
      exit(1);
     }
    if (!quiet)
      printf("Writing␣`%s'␣", output_file);

    write_pre();
    write_data();
    write_post();
    fclose(out);

    if (!quiet)
      printf("\n");
   }
  else
    write_data();
 }
```

13.

The preamble has two bytes at the very beginning, PRE and GF_ID. PRE starts the preamble, and GF_ID is the Generic Font Identity Byte. The next bytes are a string in Pascal format containing a header, the date, and the time. Strings in Pascal format start with the length of the string and have no terminating NULL byte.

```
#define GF_ID 131
#define PRE 247
#define header " hbf2gf output "
```

⟨ Prototypes 11 ⟩ +≡

```
void write_pre(void);
```

14.

```
void write_pre(void)
{char out_s[40], s[20];
  time_t secs_now;
  struct tm *time_now;

  strcpy(out_s, header);

  secs_now = time(NULL);                                   /* system date and time */
  time_now = localtime(&secs_now);
  strftime(s, 20, "%Y.%m.%d:%H.%M", time_now);
  strcat(out_s, s);

  fputc(PRE, out);
  fputc(GF_ID, out);
  fputc(strlen(out_s), out);
  fputs(out_s, out);
}
```

15.

write_data( ) produces the middle part of the GF file. It first sets *char_adr_p* equal to the address of *char_adr*[ ] which will contain file offsets of the compressed characters.

input_size_x and *input_size_y* reflect the original dimensions of the bitmap font, *pk_output_size_x* and *pk_output_size_y* contain the width and height of the output character box (in pixels), *pk_offset_x* and *pk_offset_y* define the baseline of the font. The same names starting with '*tfm_*' instead of '*pk_*' are used for TFM files (values are multiples of design size). *mag_x* and *mag_y* hold the scaling factors which are needed to reach *design_size*. *slant* defines the slant ($\Delta x/\Delta y$), *target_size_x* and *target_size_y* will be the final dimensions; *magstep_x* = *target_size_x* / *design_size* is TeX's \magstep.

The C standard specifies that all global values will be automatically set to 0 if no initialization value is given.

⟨ Global variables 2 ⟩ +≡

```
long char_adr[256];
long *char_adr_p;

int pk_offset_x;   /* horizontal offset (increase character width a bit; will be applied on both the left
    and the right side) */
double tfm_offset_x;
int pk_offset_y;                      /* vertical offset (must be configured to desired font size) */
double tfm_offset_y;
```

```
int input_size_x;
int input_size_y;
const char *font_encoding;                    /* taken from the HBF file */
int pk_width;                                 /* without slant */
int pk_output_size_x;   /* the output character box dimensions without offsets */
double tfm_output_size_x;
int pk_output_size_y;
double tfm_output_size_y;

double design_size = 10.0;                    /* in points */
double target_size_x;                         /* in points */
double target_size_y;
double magstep_x;
double magstep_y;
double slant;
int rotation;

double mag_x;          /* horizontal and vertical magnification values */
double mag_y;

int empty_char;   /* a flag whether the character does not exist or is empty */
int last_char;                /* the last valid character in a GF file */

int dot_count;
   /* this counts the processed characters; every ten characters a dot is output to the screen */
```

16.

⟨ Prototypes 11 ⟩ +≡

```
void write_data(void);
```

17.

```
void write_data(void)
{ dot_count = 0;
  char_adr_p = char_adr;
  for (last_char = 0; (last_char < 256) ∧ ! end_of_file; last_char ++)
    ⟨ Write character 18 ⟩
}
```

18.

The code in this section saves the current file position first and calls *make_pixel_array*( ), which expands and scales the character bitmap.

`BOC` (and `BOC1`), the Begin Of Character command byte, must be followed by the character code and the dimensions of the character as explained in "METAFONT—the program" (corrected by vertical and horizontal offsets).

write_coding( ) compresses and outputs the bitmap; `EOC` (End Of Character) finishes the current character.

```
#define BOC 67
#define BOC1 68                          /* simplified version of BOC */
#define EOC 69

⟨ Write character 18 ⟩ ≡
  {if (dot_count++ % 10 ≡ 0)             /* a progress report for impatient users */
      if (pk_files ∧ !quiet)
        {printf(".");
         fflush(stdout);
        }

    empty_char = FALSE;
    make_pixel_array( );
    if (end_of_file)
      return;

    if (pk_files)
     {*char_adr_p = ftell(out);
      char_adr_p++;

      if (empty_char)
        {fputc(BOC1, out);
         fputc((unsigned char) last_char, out);
         fputc(0, out);
         fputc(0, out);
         fputc(0, out);
         fputc(0, out);
         fputc(EOC, out);
        }
      else
        {fputc(BOC, out);
         fputl(last_char, out);
         fputl(−1L, out);
         fputl(pk_offset_x, out);
         fputl(pk_output_size_x + pk_offset_x, out);
         fputl(pk_offset_y, out);
         fputl(pk_output_size_y + pk_offset_y, out);

         write_coding( );

         fputc(EOC, out);
        }
     }
  }
```

This code is used in section 17.

19.

The current GF file will be completed with data written by *write_post*( ). The end consists of three sections: "special", "post", and "postpost". The first contains material not used by TeX itself but which can be used by other programs like GFtoDVI or for documentary purposes (*coding*[ ] and *comment*[ ]). The second describes the font as a whole, and the last marks the end of the file.

pk_total_min_x up to *pk_total_max_y* define the greatest bounding box of this file (including offsets); the horizontal character escapement after drawing the character is *pk_dx*. *tfm_width* is the width in multiples of the design size ignoring the target size.

```
#define _2_16 65536.0      /* 2^16 */
#define _2_20 1048576.0    /* 2^20 */

⟨ Global variables 2 ⟩ +≡
  char coding[STRING_LENGTH + 1];    /* a comment describing the font encoding */
  char comment[STRING_LENGTH + 1];   /* a comment describing the font */

  unsigned long checksum;

  long pk_total_min_x;
  long pk_total_max_x;
  long pk_total_min_y;
  long pk_total_max_y;

  int dpi_x;        /* printer resolution */
  int dpi_y;
  double ppp_x;     /* pixels per point */
  double ppp_y;
```

20.

To clarify the meaning of these values see the sections about the metrics and configuration file also.

TeX defines that 72.27 points are exactly 1 inch.

```
⟨ Prototypes 11 ⟩ +≡
  void write_post(void);
```

21.

```
  void write_post(void)
  {long special_adr;
   long post_adr;

   long designsize = design_size * _2_20;    /* design size *2^20 */

   int pk_dx;
   long tfm_width;

   int i;
   long temp;

   ppp_x = dpi_x/72.27 * magstep_x;
   ppp_y = dpi_y/72.27 * magstep_y;

   pk_total_min_x = pk_offset_x;
   pk_total_max_x = pk_output_size_x + 2 * pk_offset_x;
   pk_total_min_y = pk_offset_y;
   pk_total_max_y = pk_output_size_y + pk_offset_y;
```

```
  pk_dx = pk_width + 2 * pk_offset_x;                              /* no slant */
  tfm_width = (tfm_output_size_x + 2 * tfm_offset_x) * _2_20;
                                   /* width in multiples of design size *2^20 */
  ⟨ Special section 22 ⟩
  ⟨ Post section 23 ⟩
  ⟨ Postpost section 24 ⟩
 }
```

22.

XXXn will be followed by n bytes representing the length of a string which follows immediately. YYY is a 32 bit integer which is normally connected with the preceding string (but not used here). *special_adr* contains the address of the "special section". All items here are optional.

```
#define XXX1 239                       /* these are all special command bytes */
#define XXX2 240                                                /* not used */
#define XXX3 241                                                /* not used */
#define XXX4 242                                                /* not used */
#define YYY 243                                                 /* not used */

⟨ Special section 22 ⟩ ≡
  special_adr = ftell(out);
  if (*coding)
   {fputc(XXX1, out);                      /* XXX1 implies a string length < 256 */
    fputc(strlen(coding), out);
    fputs(coding, out);
   }
  if (*comment)
   {fputc(XXX1, out);
    fputc(strlen(comment), out);
    fputs(comment, out);
   }
```

This code is used in section 21.

23.

All character offsets collected in *char_adr* will be written to the output file. *fputl*( ) writes a 32 bit integer into a file.

CHAR_LOC0 (and CHAR_LOC) is the first byte of a character locator (i.e., offset, character code, and width information). POST starts the postamble, and *post_adr* points to the beginning byte of the postamble.

```
#define POST 248

#define CHAR_LOC 245
#define CHAR_LOC0 246                         /* simplified version of CHAR_LOC */

⟨ Post section 23 ⟩ ≡
  post_adr = ftell(out);
  fputc(POST, out);
  fputl(special_adr, out);
```

```
fputl(designsize, out);
fputl(checksum, out);
fputl(ppp_x * _2_16, out);
fputl(ppp_y * _2_16, out);
fputl(pk_total_min_x, out);
fputl(pk_total_max_x, out);
fputl(pk_total_min_y, out);
fputl(pk_total_max_y, out);
char_adr_p = char_adr;
if (pk_dx < 256)
 {for (i = 0; i < last_char; i++)                 /* the character locators */
   {fputc(CHAR_LOC0, out);
    fputc(i, out);
    fputc(pk_dx, out);
    fputl(tfm_width, out);
    fputl(*char_adr_p++, out);
   }
 }
else                                   /* will only happen if MAX_CHAR_SIZE ≥ 256 */
 {for (i = 0; i < last_char; i++)
   {fputc(CHAR_LOC, out);
    fputc(i, out);
    fputl(pk_dx * _2_16, out);
    fputl(0, out);
    fputl(tfm_width, out);
    fputl(*char_adr_p++, out);
   }
 }
```

This code is used in section 21.

24.

POSTPOST starts the section after the postamble. To get all information in a GF file, you must start here. The very last bytes of the file have the value POSTPOST_ID (the file is filled with at least 4 of these bytes until a file length of a multiple of 4 is reached). Going backwards a GF_ID will be next, then comes the address of the postamble section.

Jumping to the postamble, a POST byte comes first, then the address of the special section, and afterwards all character offsets. These offsets and addresses describe the whole file.

```
#define POSTPOST 249
#define POSTPOST_ID 223
```

⟨ Postpost section 24 ⟩ ≡

```
fputc(POSTPOST, out);
fputl(post_adr, out);
fputc(GF_ID, out);
temp = ftell(out);
i = (int) (temp % 4) + 4;
while (i--)
  fputc(POSTPOST_ID, out);
```

This code is used in section 21.

25.

TeX wants the most significant byte first.

⟨ Prototypes 11 ⟩ +≡
void *fputl*(**long**, **FILE** ∗);

26.

```
void fputl(num, f)
    long num;
    FILE *f;
{fputc(num >> 24, f);
 fputc(num >> 16, f);
 fputc(num >> 8, f);
 fputc(num, f);
}
```

27.

make_pixel_array( ) scales a character into the array *out_char*[ ] where each *byte* represents one pixel, contrary to the input file where each *bit* is used to store the character bitmap. `BLACK` indicates a black pixel.

The scaling routine was modeled after the program `pnmscale` of the `pbmplus` package. `pbmplus` was designed to handle arbitrary pictures, and bitmaps are only a special case of a graymap with values from 0 for white up to `PIXEL_MAXVAL` = 255 for black.

If `EOF` is encountered, *end_of_file* is set and the function returns immediately.

```
#define BLACK 1
#define WHITE 0

#define PIXEL_MAXVAL 255
#define SCALE 4096
#define HALFSCALE 2048

#define MAX_CHAR_SIZE 1023
```

⟨ Global variables 2 ⟩ +≡

```
HBF_CHAR code;
const unsigned char *bitmap;
        /* a proper input bitmap array will be allocated by the HBF API */
unsigned char *bP;

unsigned char out_char[MAX_CHAR_SIZE * MAX_CHAR_SIZE + 1];   /* the output bitmap array */
unsigned char *out_char_p;

unsigned char pixelrow[MAX_CHAR_SIZE];
unsigned char temp_pixelrow[MAX_CHAR_SIZE];
unsigned char new_pixelrow[MAX_CHAR_SIZE + 1];   /* we need space to append a white pixel */

int curr_row;        /* for read_row( ) if we access the glyph rotated */
long grayrow[MAX_CHAR_SIZE];
```

long s_mag_x, s_mag_y, s_slant;

28.

We need to initialize the *grayrow*[ ] array together with some other variables.

Two steps are necessary to compute the *code* if we are in the METAFONT-like mode. Assuming that we search the code `0xXXYY`, we first set *code* to the value `0xXXAA`, where `0xAA` is equal to *min_2_byte* (getting *offset* as the number of remaining characters to reach the first character in our given subfont), then we increment *code* (and decrement *offset*) until *offset* equals 0.

```
⟨ Initialize variables 28 ⟩ ≡
  { int col, offset;

    if (rotation)
    { int tmp;

      tmp = input_size_x;
      input_size_x = input_size_y;
      input_size_y = tmp;
    }
    if (mf_like)
    { target_size_x = design_size * (x_resolution / dpi_x);
      target_size_y = design_size * (x_resolution * y_scale / dpi_y);
    }
    else
      target_size_x = target_size_y = design_size;
    magstep_x = target_size_x / design_size;
    magstep_y = target_size_y / design_size;

    pk_offset_x = offset_x * magstep_x + 0.5;
    pk_offset_y = offset_y * magstep_y + 0.5;
    tfm_offset_x = offset_x / (dpi_x / 72.27) / design_size;
    tfm_offset_y = offset_y / (dpi_y / 72.27) / design_size;

    pk_width = input_size_x * mag_x * magstep_x + 0.5;        /* without slant */
    pk_output_size_x = input_size_x * mag_x * magstep_x + input_size_y * mag_y * magstep_y * slant + 0.5;
    pk_output_size_y = input_size_y * mag_y * magstep_y + 0.5;
    tfm_output_size_x = input_size_x * mag_x / (dpi_x / 72.27) / design_size;
    tfm_output_size_y = input_size_y * mag_y / (dpi_y / 72.27) / design_size;
    if (pk_output_size_x > MAX_CHAR_SIZE)
    { fprintf(stderr, "Output␣character␣box␣width␣too␣big\n");
      exit(1);
    }
    if (pk_output_size_y > MAX_CHAR_SIZE)
    { fprintf(stderr, "Output␣character␣box␣height␣too␣big\n");
      exit(1);
    }

    for (col = 0; col < input_size_x; ++col)
      grayrow[col] = HALFSCALE;

    if (!mf_like)
      code = (min_char & #FF00) + min_2_byte;
    else
    { if ((file_number < (unicode ? 0 : 1)) ∨ (file_number ≥ #100))
      { fprintf(stderr, "Invalid␣subfile␣number\n");
        exit(1);
      }
```

```
    if (unicode)
      {offset = 0;
      code = file_number * #100;
      }
    else
      {offset = (file_number - 1) * 256 % nmb_2_bytes;
      code = (min_char & #FF00) + min_2_byte + (file_number - 1) * 256/nmb_2_bytes * #100;
      }
    while (offset--)
      while (! b2_codes[code++ & #FF])          /* eliminate invalid b2_codes */
        ;
    if (code > max_char)
      {fprintf(stderr, "Invalid␣subfile␣number\n");
      exit(1);
      }
  }
  s_mag_x = mag_x * magstep_x * SCALE;
  s_mag_y = mag_y * magstep_y * SCALE;
  s_slant = slant * SCALE;
}
```

This code is used in section 4.

29.

All arrays of the *pixelrow* family contain gray values. While scaling with non-integer values a pixel of the input bitmap will normally not align with the pixel grid of the output bitmap (geometrically spoken). In this case we first compute the fractions of input pixel rows scaled vertically and add the corresponding gray values until a temporary row is produced. Then we repeat this procedure horizontally pixel by pixel and write the result into an output array.

```
⟨Prototypes 11⟩ +≡
  void make_pixel_array(void);
```

30.

```
void make_pixel_array(void)
{unsigned char *prP;
  unsigned char *temp_prP;
  unsigned char *new_prP;
  long *grP;

  register unsigned char *xP;
  register unsigned char *nxP;

  register int row, col;
  int rows_read = 0;
  register int need_to_read_row = 1;

  long frac_row_to_fill = SCALE;
  long frac_row_left = s_mag_y;

  int no_code = FALSE;
```

```
    prP = pixelrow;
    temp_prP = temp_pixelrow;
    new_prP = new_pixelrow;
    grP = grayrow;
    out_char_p = out_char;                    /* will be increased by write_row() */
again:
    if (b2_codes[code & #FF])                 /* a valid second byte? */
     {if (pk_files)
        {bitmap = hbfGetBitmap(hbf, code);
         bP = (unsigned char *) bitmap;       /* will be increased by read_row() */
         if (!bitmap)
            empty_char = TRUE;
         else
            ⟨Scale row by row 31⟩
        }
     }
    else
      no_code = TRUE;
    if ((code & #FF) ≡ max_2_byte)
      code += #FF − (max_2_byte − min_2_byte);   /* go to next plane */
    if (code ≥ max_char)
     {end_of_file = TRUE;
      return;
     }
    code++;
    if (no_code)
     {no_code = FALSE;
      goto again;
     }
  }
```

31.

```
⟨Scale row by row 31⟩ ≡
  {if (pk_output_size_y ≡ input_size_y)     /* shortcut Y scaling if possible */
      temp_prP = prP;
    curr_row = input_size_y − 1;            /* only needed for rotated glyphs */
    for (row = 0; row < pk_output_size_y; ++row)
     {⟨Scale Y from pixelrow[] into temp_pixelrow[] 32⟩
      ⟨Scale X from temp_pixelrow[] into new_pixelrow[] and write it into out_char[] 34⟩
     }
  }
```

This code is used in section 30.

32.

⟨ Scale Y from *pixelrow*[ ] into *temp_pixelrow*[ ] 32 ⟩ ≡

```
if (pk_output_size_y ≡ input_size_y)      /* shortcut Y scaling if possible */
  read_row(prP);
else
 {while (frac_row_left < frac_row_to_fill)
   {if (need_to_read_row)
      if (rows_read < input_size_y)
       {read_row(prP);
        ++rows_read;
       }
    for (col = 0, xP = prP; col < input_size_x; ++col, ++xP)
      grP[col] += frac_row_left * (*xP);
    frac_row_to_fill −= frac_row_left;
    frac_row_left = s_mag_y;
    need_to_read_row = 1;
   }
  ⟨ Produce a temporary row 33 ⟩
 }
```

This code is used in section 31.

33.

Now $frac\_row\_left \geq frac\_row\_to\_fill$, so we can produce a row.

⟨ Produce a temporary row 33 ⟩ ≡

```
if (need_to_read_row)
  if (rows_read < input_size_y)
   {read_row(prP);
    ++rows_read;
    need_to_read_row = 0;
   }
for (col = 0, xP = prP, nxP = temp_prP; col < input_size_x; ++col, ++xP, ++nxP)
 {register long g;
  g = grP[col] + frac_row_to_fill * (*xP);
  g /= SCALE;
  if (g > PIXEL_MAXVAL)
    g = PIXEL_MAXVAL;
  *nxP = g;
  grP[col] = HALFSCALE;
 }
frac_row_left −= frac_row_to_fill;
if (frac_row_left ≡ 0)
 {frac_row_left = s_mag_y;
  need_to_read_row = 1;
 }
frac_row_to_fill = SCALE;
```

This code is used in section 32.

34.

To implement the slant we move the starting point nxP to the right according to the corresponding y value. To simplify life only positive shift values are allowed.

We always append a white pixel to avoid artefacts at the end of the line produced by the last line. This rule sets the second condition that the slant must not be greater than 1—such a large slant would be unusable anyway for typesetting purposes.

⟨ Scale X from *temp_pixelrow*[ ] into *new_pixelrow*[ ] and write it into *out_char*[ ] 34 ⟩ ≡

```
if (pk_width ≡ input_size_x ∧ s_slant ≡ 0)          /* shortcut X scaling if possible */
  write_row(temp_prP);
else
 {register long g = HALFSCALE;
  register long frac_col_to_fill = SCALE;
  register long frac_col_left;
  register int need_col = 0;

  nxP = new_prP;

  frac_col_left = (pk_output_size_y − row) ∗ s_slant;
  while (frac_col_left ≥ frac_col_to_fill)
   {∗(nxP ++) = 0;
    frac_col_left −= frac_col_to_fill;
   }

  if (frac_col_left > 0)
    frac_col_to_fill −= frac_col_left;

  for (col = 0, xP = temp_prP; col < input_size_x; ++col, ++xP)
   {frac_col_left = s_mag_x;
    while (frac_col_left ≥ frac_col_to_fill)
     {if (need_col)
        {++nxP;
         g = HALFSCALE;
        }

      g += frac_col_to_fill ∗ (∗xP);
      g /= SCALE;
      if (g > PIXEL_MAXVAL)
        g = PIXEL_MAXVAL;

      ∗nxP = g;
      frac_col_left −= frac_col_to_fill;
      frac_col_to_fill = SCALE;
      need_col = 1;
     }
    if (frac_col_left > 0)
     {if (need_col)
        {++nxP;
         g = HALFSCALE;
         need_col = 0;
        }

      g += frac_col_left ∗ (∗xP);
      frac_col_to_fill −= frac_col_left;
     }
   }
```

```
    ⟨ Write out a row 35 ⟩
  }
```

This code is used in section 31.

35.

```
⟨ Write out a row 35 ⟩ ≡
  if (frac_col_to_fill > 0)
   { --xP;
     g += frac_col_to_fill * (*xP);
   }
  if (!need_col)
   { g /= SCALE;
     if (g > PIXEL_MAXVAL)
        g = PIXEL_MAXVAL;
     *nxP = g;
   }
  *(++nxP) = 0;                                    /* append a blank pixel */
  write_row(new_prP);
```

This code is used in section 34.

36.

read_row( ) reads a row from *bitmap*[ ] and converts it into a graymap row. If the *rotation* flag has been set, we get the proper column instead (note that in this case *input_size_x* already reflects the width of the rotated glyph).

```
⟨ Prototypes 11 ⟩ +≡
#ifdef __GNUC__
  __inline__
#endif
  void read_row(unsigned char *);
```

37.

```
#ifdef __GNUC__
  __inline__
#endif
  void read_row(pixelrow)
      unsigned char *pixelrow;
   {register int col, bitshift, offset;
    register unsigned char *xP;
    register unsigned char item = 0;

    if (rotation)
     {bitshift = 7 − (curr_row % 8);
      offset = (input_size_y + 7)/8;
      bP = (unsigned char *) bitmap + curr_row/8;
      for (col = 0, xP = pixelrow; col < input_size_x; ++col, ++xP)
       {*xP = ((*bP >> bitshift) & 1) ≡ 1 ? PIXEL_MAXVAL : 0;
        bP += offset;
       }
      curr_row −−;
     }
    else
     {bitshift = −1;
      for (col = 0, xP = pixelrow; col < input_size_x; ++col, ++xP)
       {if (bitshift ≡ −1)
         {item = *(bP++);                    /* increase input bitmap pointer */
          bitshift = 7;
         }
        *xP = ((item >> bitshift) & 1) ≡ 1 ? PIXEL_MAXVAL : 0;
        −−bitshift;
       }
     }
   }
```

38.

write_row( ) converts the graymap back into a bitmap using a simple threshold.

```
⟨Global variables 2⟩ +≡
  int threshold = 128;
```

39.

```
⟨Prototypes 11⟩ +≡
#ifdef __GNUC__
  __inline__
#endif
  void write_row(unsigned char *);
```

40.

```
#ifdef __GNUC__
  __inline__
#endif
  void write_row(pixelrow)
       unsigned char *pixelrow;
   {register int col;
    register unsigned char *xP;

    for (col = 0, xP = pixelrow; col < pk_output_size_x; ++col, ++xP)
       *(out_char_p++) = (*xP >= threshold) ? 1 : 0;        /* increase output bitmap pointer */
   }
```

41.

Now comes the most interesting routine. The pixel array will be compressed in sequences of black and white pixels.

SKIP0, SKIP1, and SKIP2 indicate how many blank lines will be skipped. PAINT_(x) means that the next x pixels will have the same color, then the color changes. NEW_ROW_(x) is the first black pixel in the next row.

An example: the pixel sequence 111100011001 [new row] 000111011110 will be output as 4 3 2 2 1 77 3 1 4 1.

Commands with an ending 'n' in its name indicate that the next n bytes should be read as the counter. Example: SKIP1 26 means 'skip the next 26 rows'.

For further details please refer to "METAFONT—the program".

```
#define PAINT_(x) (x)                    /* 0 <= x <= 63 */
#define PAINT1 64
#define PAINT2 65
#define PAINT3 66                        /* not used */

#define SKIP0 70
#define SKIP1 71
#define SKIP2 72
#define SKIP3 73                         /* not used */

#define NEW_ROW_(x) ((x) + 74)           /* 0 <= x <= 164 */

#define NOOP 244                         /* not used */
```

⟨ Prototypes 11 ⟩ +≡

```
  void write_coding(void);
```

42.

The **goto** *start* instruction causes some compilers to complain about "Unreachable code ..." or something similar.

```
  void write_coding(void)
   {register int count, skip;
    register unsigned char paint;
    register int x, y;
    register unsigned char *cp;
```

$x = 0;$
$y = 0;$
$cp = out\_char + y * pk\_output\_size\_x + x;$
$count = skip = 0;$
$paint =$ `WHITE`;
goto *start*;

while $(y < pk\_output\_size\_y)$
{⟨ Search blank lines 43 ⟩
start:
⟨ Process rest of line 44 ⟩
y++;
}
}

43.

⟨ Search blank lines 43 ⟩ ≡

```
count = 0;
x = 0;
cp = out_char + y * pk_output_size_x + x;
while (x < pk_output_size_x)
 {if (*cp ≡ paint)
     count++;
   else
   {if (skip ≡ 0)
     {if (count ≤ 164)
         fputc(NEW_ROW_(count), out);
       else
       {fputc(SKIP0, out);
         if (count < 256)
          {fputc(PAINT1, out);
           fputc(count, out);
          }
         else
          {fputc(PAINT2, out);
           fputc(count ≫ 8, out);
           fputc(count & #FF, out);
          }
       }
     }
     else
     {if (skip ≡ 1)
         fputc(SKIP0, out);
       else
       {if (skip < 256)
          {fputc(SKIP1, out);
           fputc(skip, out);
          }
         else
          {fputc(SKIP2, out);
           fputc(skip ≫ 8, out);
           fputc(skip & #FF, out);
          }
       }
       skip = 0;
       if (count < 64)
         fputc(PAINT_(count), out);
       else if (count < 256)
        {fputc(PAINT1, out);
         fputc(count, out);
        }
       else
        {fputc(PAINT2, out);
         fputc(count ≫ 8, out);
         fputc(count & #FF, out);
        }
```

```
          }
          count = 0;
          paint = BLACK;
          break;
        }
      x++;
      cp++;
    }
  if (x ≥ pk_output_size_x)
    {skip++;
      y++;
      continue;
    }
```

This code is used in section 42.

44.

⟨ Process rest of line 44 ⟩ ≡

```
while (x < pk_output_size_x)
  {if (*cp ≡ paint)
      count++;
    else
     {if (count < 64)
         fputc(PAINT_(count), out);
       else if (count < 256)
        {fputc(PAINT1, out);
         fputc(count, out);
        }
       else
        {fputc(PAINT2, out);
         fputc(count ≫ 8, out);
         fputc(count & #FF, out);
        }
       count = 1;
       paint = BLACK − paint;
     }
    x++;
    cp++;
  }
if (paint ≡ BLACK)
  {if (count < 64)
      fputc(PAINT_(count), out);
    else if (count < 256)
     {fputc(PAINT1, out);
      fputc(count, out);
     }
    else
     {fputc(PAINT2, out);
      fputc(count ≫ 8, out);
      fputc(count & #FF, out);
     }
    paint = WHITE;
  }
```

This code is used in section 42.

45. The font metrics file.

This routine creates one PL file with the font properties. None of the TeX font dimensions are needed because you never will use CJK fonts directly, and intercharacter stretching is handled by the *CJK* macro `\CJKglue`. (Other packages may define similar commands.)

The name of the PL file will contain the running two digits in METAFONT-like mode only.

It makes sense *not* to compute the check sum automatically for two reasons. Firstly, since TeX's checksum algorithm is based on the character width, the number of valid characters, and the designsize, there is a much higher chance that two subfonts from different HBF fonts have the same check sum than it is for ordinary fonts, because all characters have the same width, usually 256 characters in a subfont, and very often the same design size. Secondly, and this is more important, we create just one TFM file for all subfonts regardless of the real number of characters in a particular subfont.

To have an identification string in the TFM file, we split it into single bytes and use the HEADER command repeatedly.

⟨ Prototypes 11 ⟩ +≡
  void *write_pl*(**void**);

46.

```
void write_pl(void)
 {int i, pos;
  char output_file[FILE_NAME_LENGTH + 1];
  long t, sc;
  char *s;
  char tfm_header[] = "Created␣by␣hbf2gf";

  file_number--;                                   /* for METAFONT-like mode */

  if (mf_like)
   {if (unicode)
       sprintf(output_file, "%s%02x.pl", output_name, file_number);
    else
       sprintf(output_file, "%s%02i.pl", output_name, file_number);
   }
  else
    sprintf(output_file, "%s.pl", output_name);

  if (!(out = fopen(output_file, WRITE_TXT)))
   {fprintf(stderr, "Couldn't␣open␣`%s'\n", output_file);
    exit(1);
   }
  if (!quiet)
    printf("\nWriting␣`%s'\n", output_file);

  fprintf(out,
  "\n(FAMILY␣%s%d)"
  "\n(CODINGSCHEME␣CJK-%s)", output_name, file_number, font_encoding);

  fprintf(out,
  "\n(DESIGNSIZE␣R␣%.6f)"
  "\n(COMMENT␣DESIGNSIZE␣IS␣IN␣POINTS)"
  "\n(COMMENT␣OTHER␣SIZES␣ARE␣MULTIPLES␣OF␣DESIGNSIZE)"
  "\n(CHECKSUM␣O␣%lo)"
  "\n(FONTDIMEN"
  "\n␣␣␣(SLANT␣R␣%.6f)"
  "\n␣␣␣(SPACE␣R␣0.0)"
  "\n␣␣␣(STRETCH␣R␣0.0)"
  "\n␣␣␣(SHRINK␣R␣0.0)"
  "\n␣␣␣(XHEIGHT␣R␣1.0)"
  "\n␣␣␣(QUAD␣R␣1.0)"
  "\n␣␣␣(EXTRASPACE␣R␣0.0)"
  "\n␣␣␣)", design_size, checksum, slant);

  s = tfm_header;
  i = strlen(s);
  t = ((long) i) << 24;
  sc = 16;
  pos = 18;
```

```
fprintf(out, "\n");
while (i > 0)
  {t |= ((long) (*(unsigned char *) s++)) << sc;
   sc -= 8;
   if (sc < 0)
     {fprintf(out, "\n(HEADER␣D␣%d␣O␣%lo)", pos, t);
      t = 0;
      sc = 24;
      pos++;
     }
   i--;
  }
if (t)
   fprintf(out, "\n(HEADER␣D␣%d␣O␣%lo)", pos, t);
fprintf(out, "\n");
for (i = 0; i < 256; i++)
  {fprintf(out,
   "\n(CHARACTER␣O␣%o"
   "\n␣␣␣(CHARWD␣R␣%.6f)"
   "\n␣␣␣(CHARHT␣R␣%.6f)"
   "\n␣␣␣(CHARDP␣R␣%.6f)"
   "\n␣␣␣(CHARIC␣R␣%.6f)"
   "\n␣␣␣)",
   i, tfm_output_size_x + 2 * tfm_offset_x, tfm_output_size_y + tfm_offset_y, -tfm_offset_y,
        slant * (tfm_output_size_y + tfm_offset_y));
  }
 fclose(out);
}
```

47. The extended virtual font file for Ω.

The following is very similar to *write_pl*( ); we simply map the glyphs of the subfonts back to the original encoding positions.

```
⟨ Prototypes 11 ⟩ +≡
  void write_ovp(void);
```

48.

```
void write_ovp(void)
{int c, i, nmb_subfonts, remainder, count, pos;
  char output_file[FILE_NAME_LENGTH + 1];
  long t, sc;
  char *s;
  char ofm_header[] = "Created␣by␣hbf2gf";

  nmb_subfonts = ((max_char − (min_char & #FF00))/256 * nmb_2_bytes)/256 + 1;
  remainder = ((max_char − (min_char & #FF00))/256 * nmb_2_bytes) % 256;
                                /* correction for the last incomplete second byte range */
  for (count = 0; count < (max_char & #FF); count++)
    if (b2_codes[count])
      remainder++;
  if (remainder ≥ 256)
    nmb_subfonts++;

  sprintf(output_file, "%s.ovp", output_name);

  if (!(out = fopen(output_file, WRITE_TXT)))
   {fprintf(stderr, "Couldn't␣open␣'%s'\n", output_file);
    exit(1);
   }
  if (!quiet)
    printf("\nWriting␣'%s'\n", output_file);

  fprintf(out,
  "\n(VTITLE␣Omega␣virtual␣font␣created␣by␣hbf2gf)"
  "\n(DESIGNSIZE␣R␣%.6f)"
  "\n(COMMENT␣DESIGNSIZE␣IS␣IN␣POINTS)"
  "\n(COMMENT␣OTHER␣SIZES␣ARE␣MULTIPLES␣OF␣DESIGNSIZE)"
  "\n(CHECKSUM␣O␣%lo)"
  "\n(FONTDIMEN"
  "\n␣␣␣(SLANT␣R␣%.6f)"
  "\n␣␣␣(SPACE␣R␣0.0)"
  "\n␣␣␣(STRETCH␣R␣0.0)"
  "\n␣␣␣(SHRINK␣R␣0.0)"
  "\n␣␣␣(XHEIGHT␣R␣1.0)"
  "\n␣␣␣(QUAD␣R␣1.0)"
  "\n␣␣␣(EXTRASPACE␣R␣0.0)"
  "\n␣␣␣)", design_size, checksum, slant);

  s = ofm_header;
  i = strlen(s);
  t = ((long) i) ≪ 24;
  sc = 16;
  pos = 18;
```

```
fprintf(out, "\n");
while (i > 0)
  {t |= ((long) (*(unsigned char *) s++)) << sc;
   sc -= 8;
   if (sc < 0)
     {fprintf(out, "\n(HEADER␣D␣%d␣O␣%lo)", pos, t);
      t = 0;
      sc = 24;
      pos++;
     }
   i--;
  }
if (t)
   fprintf(out, "\n(HEADER␣D␣%d␣O␣%lo)", pos, t);
fprintf(out, "\n");
for (i = 0; i < nmb_subfonts; i++)
  {fprintf(out,
   "\n(MAPFONT␣D␣%i"
   "\n␣␣␣(FONTNAME␣%s%02i)"
   "\n␣␣␣(FONTCHECKSUM␣O␣%lo)"
   "\n␣␣␣(FONTAT␣R␣1.0)"
   "\n␣␣␣(FONTDSIZE␣R␣%.6f)"
   "\n␣␣␣)", i, output_name, i + 1, checksum, design_size);
  }
for (c = min_char, i = 0, count = 0; c ≤ max_char; c++)
  {if (b2_codes[c & #FF] ≡ VALID_SUBCODE)
     {fprintf(out,
      "\n(CHARACTER␣O␣%o"
      "\n␣␣␣(CHARWD␣R␣%.6f)"
      "\n␣␣␣(CHARHT␣R␣%.6f)"
      "\n␣␣␣(CHARDP␣R␣%.6f)"
      "\n␣␣␣(CHARIC␣R␣%.6f)"
      "\n␣␣␣(MAP"
      "\n␣␣␣␣␣␣(SELECTFONT␣D␣%i)"
      "\n␣␣␣␣␣␣(SETCHAR␣O␣%o)"
      "\n␣␣␣␣␣␣)"
      "\n␣␣␣)",
      c, tfm_output_size_x + 2 * tfm_offset_x, tfm_output_size_y + tfm_offset_y, −tfm_offset_y,
            slant * (tfm_output_size_y + tfm_offset_y), i, count);
      count++;
      if (count ≡ 256)
        {count = 0;
         i++;
        }
     }
   else
      continue;
  }
fclose(out);
}
```

49. The job file.

This routine is the most system specific one. If your operating system needs a different outline, make appropriate changes here.

You have to call this batch file after `hbf2gf` has finished (if not in METAFONT-like mode). It will transform the `GF` files into `PK` files and delete the now unnecessary `GF` files, then transform the `PL` file into a `TFM` file and copy it *nmb_files* times. The name of the job file is *output_name*.

```
#define EXTENSION_LENGTH 8    /* the maximal length of a file extension */

#define GFTOPK_NAME "gftopk"
#define PLTOTF_NAME "pltotf"
#define OVP2OVF_NAME "ovp2ovf"
```

⟨ Global variables 2 ⟩ +≡

```
char job_extension[EXTENSION_LENGTH + 1];
char rm_command[STRING_LENGTH + 1];
char cp_command[STRING_LENGTH + 1];
char pk_directory[STRING_LENGTH + 1];
char tfm_directory[STRING_LENGTH + 1];

int ofm_file = FALSE;
```

50.

⟨ Prototypes 11 ⟩ +≡

```
void write_job(void);
```

51.

```
void write_job(void)
{FILE *out;
  int i, j;
  char buffer[FILE_NAME_LENGTH + 1];

  strcpy(buffer, output_name);
  strcat(buffer, job_extension);
  if (!(out = fopen(buffer, WRITE_TXT)))
   {fprintf(stderr, "Couldn't open `%s'\n", buffer);
    exit(1);
   }
  if (!quiet)
    printf("\nWriting `%s'\n", buffer);

  if (pk_files)
   {if (unicode)
     {for (i = (min_char >> 8), j = 0; j < nmb_files; i++, j++)
        fprintf(out,
        "%s %s%02x.gf %s%s%02x.%.0ipk\n"
        "%s %s%02x.gf\n",
        GFTOPK_NAME, output_name, i,
        pk_directory, output_name, i, long_extension ? (int) (dpi_x * magstep_x + 0.5) : 0,
        rm_command, output_name, i);
     }
    else
     {for (i = 1; i <= nmb_files; i++)
        fprintf(out,
        "%s %s%02i.gf %s%s%02i.%.0ipk\n"
        "%s %s%02i.gf\n",
        GFTOPK_NAME, output_name, i,
        pk_directory, output_name, i, long_extension ? (int) (dpi_x * magstep_x + 0.5) : 0,
        rm_command, output_name, i);
     }
   }
  if (tfm_files)
   {fprintf(out,
    "\n"
    "%s %s.pl %s.tfm\n"
    "%s %s.pl\n"
    "\n",
    PLTOTF_NAME, output_name, output_name,
    rm_command, output_name);

    if (unicode)
     {for (i = (min_char >> 8), j = 0; j < nmb_files; i++, j++)
        fprintf(out,
        "%s %s.tfm %s%s%02x.tfm\n",
        cp_command, output_name, tfm_directory, output_name, i);
     }
    else
     {for (i = 1; i <= nmb_files; i++)
        fprintf(out,
```

```
        "%s␣%s.tfm␣%s%s%02i.tfm\n",
        cp_command, output_name, tfm_directory, output_name, i);
      }
    fprintf(out,
    "\n"
    "%s␣%s.tfm",
    rm_command, output_name);
   }
  if (ofm_file)
   {fprintf(out,
    "\n"
    "%s␣%s.ovp␣%s.ovf␣%s.ofm\n"
    "%s␣%s.ovp\n"
    "\n",
    OVP2OVF_NAME, output_name, output_name, output_name,
    rm_command, output_name);
   }
  fclose(out);
}
```

52. The configuration file.

Here is a list with all necessary keywords (and parameters):

`hbf_header`	the HBF header file name of the input font(s).
`output_name`	the name stem of the output files. Should be equal to the name of the configuration file in most cases. A running two digit decimal number starting with `01` will be appended. (For Unicode fonts see the keyword `unicode` below.)

And now all optional keywords:

`x_offset`	increases the character width. Will be applied on both sides; default is the value given in the HBF header (`HBF_BITMAP_BOUNDING_BOX`) scaled to *design*size (in pixels).
`y_offset`	shifts all characters up or down; default is the value given in the HBF header (`HBF_BITMAP_BOUNDING_BOX`) scaled to *design*size (in pixels).
`design_size`	the design size (in points) of the font. `x_offset` and `y_offset` refer to this size. Default is 10.0
`target_size`	This command is obsolete now and will be ignored.
`slant`	the slant of the font (given as $\Delta x/\Delta y$). Only values in the range $0 \le slant \le 1$ are allowed. Default is 0.0
`rotation`	if set to 'yes'. the glyphs are rotated 90 degrees counter-clockwise. The default offsets as given in the HBF header will be ignored (and set to 0). Default is 'no'.
`mag_x` `mag_y`	scaling values of the characters to reach design size. If only one magnification is given, x and y values are assumed to be equal. Default is `mag_x` = `mag_y` = 1.0
`threshold`	A value between 1 and 254 defining a threshold for converting the internal graymap into the output bitmap; lower values cut more pixels. Default value is 128.
`comment`	a comment describing the font; default is none.
`nmb_fonts`	the number of the fonts. Default value is −1 for creating all fonts.
`unicode`	if 'yes', a two digit hexadecimal number will be used as a running number, starting with the value of the first byte of the first code range. Default is 'no'.
`min_char`	the minimum of the encoding range. Specify this value if it is not identical to the lowest code value in the HBF file (to which it defaults).
`dpi_x` `dpi_y`	the horizontal and vertical resolution (in dpi) of the printer.

	If only one resolution is given, x and y values are assumed to be equal. Default is 300.
`checksum`	a checksum to identify the `GF` files with the appropriate `TFM` files. The default of this 32 bit unsigned integer is 0.
`coding`	a comment describing the coding scheme; default is none.
`pk_directory`	the destination directory of the `PK` files; default: none. Attention! The batch file will not check whether this directory exists.
`tfm_directory`	the destination directory of the `TFM` files; default: none. Attention! The batch file will not check whether this directory exists.
`pk_files`	whether to create `PK` files or not; default is 'yes'.
`tfm_files`	whether to create `TFM` files or not; default is 'yes'.
`ofm_file`	whether to create an `OFM` and an `OVF` file or not; default is 'no'.
`long_extension`	if 'yes', `PK` files will include the resolution in the extension (e.g. `gsso1201.300pk`). This affects the batch file only (default is 'yes').
`rm_command`	this shell command removes files; default: 'rm'.
`cp_command`	this shell command copies files; default: 'cp'.
`job_extension`	the extension of the batch file which calls `GFtoPK` and `PLtoTF` to convert the `GF` and the `PL` files into `PK` and `TFM` files; default is none.

The searching algorithm (for the keywords) of `hbf2gf` is case insensitive; it makes no difference whether you write for example `comment` or `CommenT`. The keywords must start a line (be in the first column), and the corresponding parameters must be on the same line with the keyword and separated by at least one space or tabulator stop. Lines starting not with a keyword are ignored.

Key values *are* case sensitive (except `yes` and `no`).

The default system dependent values are for UNIX-like operating systems; if you use for example DOS, you must write

```
long_extension   no
rm_command       del
cp_command       copy
job_extension    .bat
```

Both the values *pk_output_size_x* and *pk_output_size_y* must not exceed `MAX_CHAR_SIZE`; `x_offset` and `y_offset` are related to the design size (and not to the input size).

In METAFONT-like mode, one `GF` file and one `PL` file will be computed (depending on the command line options `-g` and `-p`), taking *x_resolution* and *y_scale* from the command line. `nmb_fonts` will always be set to 1; no job file will be created.

53.

```
#define PRINTER_MIN_RES_X 50
#define PRINTER_MIN_RES_Y 50

⟨ Global variables 2 ⟩ +≡
  char Buffer[STRING_LENGTH + 1];
```

54.

```
⟨ Prototypes 11 ⟩ +≡
  void read_config(void);
```

55.

If *config_file* isn't found in METAFONT-like mode we assume that the font isn't a HBF font at all.

```
  void read_config(void)
  { HBF_BBOX *boxp;
    char *real_config_file;

    ⟨ Handle extension 56 ⟩

    real_config_file = TeX_search_cfg_file(config_file);
    if (!real_config_file)
    { if (mf_like)
      { if (!quiet)
          printf("Couldn't find '%s'\n", config_file);
        exit(2);
      }
      else
      { fprintf(stderr, "Couldn't find '%s'\n", config_file);
        exit(1);
      }
    }
    if (!(config = fopen(real_config_file, READ_TXT)))
    { if (!testing)
      { fprintf(stderr, "Couldn't open '%s'\n", config_file);
        exit(1);
      }
      else      /* We reach this point only if no searching library is used */
      { if (!quiet)
          fprintf(stderr, "Couldn't find or open '%s'\n", config_file);
        exit(2);
      }
    }
    if (testing)
    { if (!quiet)
        printf("%s\n", real_config_file);
      exit(0);
    }
    ⟨ Necessary parameters 57 ⟩
    ⟨ Optional parameters 59 ⟩
```

```
  ⟨ Get code range 61 ⟩
  ⟨ Get sub code range 63 ⟩
  fclose(config);
}
```

56.

Here we check whether we have to add an extension.

```
⟨ Handle extension 56 ⟩ ≡
{int i, lastext = −1;
  for (i = 0; config_file[i]; i++)
    if (config_file[i] ≡ '.')
      lastext = i;
    else if (config_file[i] ≡ '/' ∨ config_file[i] ≡ ':' ∨ config_file[i] ≡ '\\')
      lastext = −1;
  if (lastext ≡ −1)
    strcat(config_file, ".cfg");
}
```

This code is used in section 55.

57.

```
⟨ Necessary parameters 57 ⟩ ≡
{char hbf_header[STRING_LENGTH + 1];
  char *real_hbf_header;
  if (!fsearch("hbf_header"))
    config_error("hbf_header");
  else
    strcpy(hbf_header, Buffer);
  real_hbf_header = TeX_search_hbf_file(hbf_header);
  if (!real_hbf_header)
   {fprintf(stderr, "Couldn't␣find␣`%s'\n", hbf_header);
    exit(1);
   }
  hbfDebug = 1;
           /* we activate error messages of the HBF API while scanning the HBF header file */
  if (!(hbf = hbfOpen(real_hbf_header)))
    exit(1);
  hbfDebug = 0;
  boxp = hbfBitmapBBox(hbf);
  input_size_x = boxp→hbf_height;                    /* will be checked later for rotation */
  input_size_y = boxp→hbf_width;
  font_encoding = hbfProperty(hbf, "HBF_CODE_SCHEME");
```

```
  if (!fsearch("output_name"))
    config_error("output_name");
  else
    strcpy(output_name, Buffer);
}
```

This code is used in section 55.

58.

```
⟨ Global variables 2 ⟩ +≡
  int offset_x;
  int offset_y;
  HBF_CHAR user_min_char;
  int have_min_char = FALSE;
```

59.

```
⟨ Optional parameters 59 ⟩ ≡
  {if (fsearch("nmb_files"))
      nmb_files = atoi(Buffer);
    if (fsearch("unicode"))
      if (Buffer[0] ≡ 'y' ∨ Buffer[0] ≡ 'Y')
        unicode = TRUE;
    if (fsearch("min_char"))
     {user_min_char = strtoul(Buffer, (char **) Λ, 0);
      have_min_char = TRUE;
     }
    if (!mf_like)
     {if (fsearch("pk_files"))
        if (Buffer[0] ≡ 'n' ∨ Buffer[0] ≡ 'N')
          pk_files = FALSE;
      if (fsearch("tfm_files"))
        if (Buffer[0] ≡ 'n' ∨ Buffer[0] ≡ 'N')
          tfm_files = FALSE;
      if (fsearch("ofm_file"))
        if (Buffer[0] ≡ 'y' ∨ Buffer[0] ≡ 'Y')
          ofm_file = TRUE;
      if (fsearch("long_extension"))
        if (Buffer[0] ≡ 'n' ∨ Buffer[0] ≡ 'N')
          long_extension = FALSE;
     }
    if (fsearch("slant"))
      slant = atof(Buffer);
    if (slant < 0.0 ∨ slant > 1.0)
     {fprintf(stderr, "Invalid␣slant\n");
      exit(1);
     }
    if (fsearch("rotation"))
      if (Buffer[0] ≡ 'y' ∨ Buffer[0] ≡ 'Y')
        rotation = TRUE;
```

if $(fsearch("mag\_x"))$
  $mag\_x = atof(Buffer);$
if $(fsearch("mag\_y"))$
  $mag\_y = atof(Buffer);$
if $(!\,mag\_x \land !\,mag\_y)$
 $\{mag\_x = 1.0;$
  $mag\_y = 1.0;$
 $\}$
if $(mag\_x \land !\,mag\_y)$
  $mag\_y = mag\_x;$
if $(mag\_y \land !\,mag\_x)$
  $mag\_x = mag\_y;$
if $(mag\_x \le 0.0)$
 $\{fprintf(stderr,$ `"Invalid␣horizontal␣magnification\n"`$);$
  $exit(1);$
 $\}$
if $(mag\_y \le 0.0)$
 $\{fprintf(stderr,$ `"Invalid␣vertical␣magnification\n"`$);$
  $exit(1);$
 $\}$
if $(fsearch("dpi\_x"))$
  $dpi\_x = atoi(Buffer);$
if $(fsearch("dpi\_y"))$
  $dpi\_y = atoi(Buffer);$
if $(!\,dpi\_x \land !\,dpi\_y)$
 $\{dpi\_x = 300;$
  $dpi\_y = 300;$
 $\}$
if $(dpi\_x \land !\,dpi\_y)$
  $dpi\_y = dpi\_x;$
if $(dpi\_y \land !\,dpi\_x)$
  $dpi\_x = dpi\_y;$
if $(dpi\_x \le$ `PRINTER_MIN_RES_X`$)$
 $\{fprintf(stderr,$ `"Invalid␣horizontal␣printer␣resolution\n"`$);$
  $exit(1);$
 $\}$
if $(dpi\_y \le$ `PRINTER_MIN_RES_Y`$)$
 $\{fprintf(stderr,$ `"Invalid␣vertical␣printer␣resolution\n"`$);$
  $exit(1);$
 $\}$
if $(fsearch("design\_size"))$
  $design\_size = atof(Buffer);$
if $(fsearch("x\_offset"))$
  $offset\_x = atoi(Buffer);$
else
  $offset\_x = rotation\ ?\ 0 : (boxp\neg hbf\_xDisplacement * mag\_x + 0.5);$
if $(fsearch("y\_offset"))$
  $offset\_y = atoi(Buffer);$
else
  $offset\_y = rotation\ ?\ 0 : (boxp\neg hbf\_yDisplacement * mag\_y + 0.5);$
if $(!\,fsearch("comment"))$

```
    comment[0] = '\0';
  else
    strcpy(comment, Buffer);
  if (fsearch("threshold"))
    threshold = atoi(Buffer);
  if (threshold ≤ 0 ∨ threshold ≥ 255)
   {fprintf(stderr, "Invalid␣threshold\n");
    exit(1);
   }
  if (!fsearch("checksum"))
    checksum = 0;
  else
    checksum = strtoul(Buffer, (char **) Λ, 0);
  if (!fsearch("coding"))
    coding[0] = '\0';
  else
    strcpy(coding, Buffer);
  if (!fsearch("pk_directory"))
    pk_directory[0] = '\0';
  else
    strcpy(pk_directory, Buffer);
  if (!fsearch("tfm_directory"))
    tfm_directory[0] = '\0';
  else
    strcpy(tfm_directory, Buffer);
  if (fsearch("rm_command"))
    strcpy(rm_command, Buffer);
  else
    strcpy(rm_command, "rm");
  if (fsearch("cp_command"))
    strcpy(cp_command, Buffer);
  else
    strcpy(cp_command, "cp");
  if (!fsearch("job_extension"))
    job_extension[0] = '\0';
  else
   {strncpy(job_extension, Buffer, EXTENSION_LENGTH);
    job_extension[EXTENSION_LENGTH] = '\0';
   }
}
```

This code is used in section 55.

60.

The function *hbfGetCodeRange*( ) is an extension to the HBF API.

Successive calls return the code ranges in ascending order; we only need the extrema of the whole code range.

In case *min_char* has been supplied in the configuration file, we use that value instead.

⟨ Global variables 2 ⟩ +≡
```
HBF_CHAR min_char, max_char;
```

61.

⟨ Get code range 61 ⟩ ≡
```
{const void *cp;
 HBF_CHAR dummy;

 cp = hbfGetCodeRange(hbf, Λ, &min_char, &max_char);
 for ( ; cp ≠ Λ; cp = hbfGetCodeRange(hbf, cp, &dummy, &max_char))
   ;

 if (have_min_char)
   min_char = user_min_char;
}
```
This code is used in section 55.

62.

The function *hbfGetByte2Range*( ) is an extension to the HBF API.

Successive calls return the byte 2 ranges in ascending order. We raise `VALID_SUBCODE` in the array *b2_codes*[ ] for all characters in subcode ranges.

```
#define VALID_SUBCODE 1
```

⟨ Global variables 2 ⟩ +≡
```
char b2_codes[256];
unsigned char min_2_byte, max_2_byte;
int nmb_2_bytes = 0;
```

63.

⟨ Get sub code range 63 ⟩ ≡
```
{const void *b2r;
 unsigned char dummy;
 int i;

 for (i = 0; i < 256; i++)
   b2_codes[i] = 0;

 b2r = hbfGetByte2Range(hbf, Λ, &min_2_byte, &max_2_byte);
 dummy = min_2_byte;
 for ( ; b2r ≠ Λ; b2r = hbfGetByte2Range(hbf, b2r, &dummy, &max_2_byte))
  {for (i = dummy; i ≤ max_2_byte; i++)
     b2_codes[i] = VALID_SUBCODE;
  }
```

```
    for (i = 0; i < 256; i++)        /* compute the number of valid b2_codes */
      if (b2_codes[i] ≡ VALID_SUBCODE)
        nmb_2_bytes++;
  }
```

This code is used in section 55.

64.

This search routine is case insignificant. Each keyword must start a line; the function checks whether the character before the keyword is a newline character (`'\n'`). It also checks the presence of a parameter and fills *Buffer* if existent. *fsearch*( ) returns 1 on success.

```
⟨ Prototypes 11 ⟩ +≡
  int fsearch(char *);
```

65.

```
  int fsearch(search_string)
      char *search_string;
  { char *P, p;
    char temp_buffer[STRING_LENGTH + 1];
    char env_name[STRING_LENGTH + 1];
    char *env_p;
    char *env_value;
    char *Buf_p;
    int Ch, ch, old_ch = '\n';
    int count = STRING_LENGTH;

    rewind(config);                                  /* we start at offset 0 */

    do
     { P = search_string;
      p = tolower(*P);
      Ch = fgetc(config);
      ch = tolower(Ch);
      while (!(ch ≡ p ∧ old_ch ≡ '\n') ∧ Ch ≠ EOF)
            /* search first character of search_string; '\n' must be the character before */
       { old_ch = ch;
        Ch = fgetc(config);
        ch = tolower(Ch);
       }

      for ( ; ; )
       { if (*(++P) ≡ '\0')
          if ((Ch = fgetc(config)) ≡ '␣' ∨ Ch ≡ '\t')
                          /* there must be a space or a tab stop after the keyword */
            goto success;
        Ch = fgetc(config);
        if (tolower(Ch) ≠ tolower(*P))
          break;
       }
     } while (Ch ≠ EOF);
```

```
    return 0;
success:
    P = temp_buffer;
    while ((Ch = fgetc(config)) ≡ '␣' ∨ Ch ≡ '\t')      /* remove leading blanks and tabs */
      ;
    while (Ch ≠ '\n' ∧ −−count > 0 ∧ Ch ≠ EOF)           /* fill Buffer */
     {*P++ = Ch;
      Ch = fgetc(config);
     }
    *P = '\0';
    if (*temp_buffer)
      ⟨ Check for environment variables 66 ⟩
    else
      *Buffer = '\0';
    return (*Buffer) ? 1 : 0;                /* is there something in the buffer? */
   }
```

66.

To make the configuration file more flexible we allow environment variables in the arguments. We scan the parameter stored in *temp_buffer* whether it contains a '`$`' character. If yes, the following code fragment tries to get an environment variable name whose value will be then fetched with *getenv*( ). An environment variable name recognized by `hbf2gf` must start with a letter or underscore; the other characters may be alphanumeric or an underscore. You can surround the environment variable name with braces to indicate where the name ends, e.g. `${FOO}`. The interpolated configuration parameter will be truncated to `STRING_LENGTH` characters. If you want to have '`$`' you must write '`$$`'.

Note that you should avoid to use such environment variables for specifying the location of the configuration file in case you have support for a file searching library like kpathsea. Its primary aim is to specify the target directories for the `pk_directory` and the `tfm_directory` keywords.

⟨ Check for environment variables 66 ⟩ ≡

```
{P = temp_buffer;
 Buf_p = Buffer;
 count = STRING_LENGTH − 1;
 while (∗P ∧ count > 0)
  {env_p = env_name;
   if (∗P ≡ '$')
    {P++;
     if (∗P ≡ '$')
      {∗(Buf_p++) = ∗(P++);
       count−−;
       continue;
      }
     while (∗P ≡ '{')
       P++;
     if (!(isalpha(∗P) ∨ ∗P ≡ '_'))
      {fprintf(stderr,
       "Invalid␣environment␣variable␣name␣in␣configuration␣file\n");
       exit(1);
      }
     ∗(env_p++) = ∗(P++);
     while (∗P)
      {if (isalnum(∗P) ∨ ∗P ≡ '_')
         ∗(env_p++) = ∗(P++);
       else
        {while (∗P ≡ '}')
           P++;
         ∗env_p = '\0';
         break;
        }
      }
     env_value = getenv(env_name);
     if (env_value)                      /∗ append the environment value to Buffer ∗/
      {while (∗env_value ∧ count > 0)
        {∗(Buf_p++) = ∗(env_value++);
         count−−;
        }
      }
    }
```

```
        else
          {*(Buf_p++) = *(P++);
           count--;
          }
      }
    *Buf_p = '\0';
  }
```

This code is used in section 65.

67.

If an error occurs, *config_error*( ) will leave the program with an error message.

```
⟨ Prototypes 11 ⟩ +≡
  void config_error(char *);
```

68.

```
  void config_error(message)
       char *message;
  { fprintf(stderr, "Couldn't find `%s' entry in configuration file\n", message);
    exit(1);
  }
```

69. File searching.

We support three searching engines: emtexdir, kpathsea, and MiKTeX (which is a Win32 port of kpathsea). For emtexdir, define `HAVE_EMTEXDIR` while compiling. For kpathsea, define `HAVE_LIBKPATHSEA`. If you have a version of kpathsea older than 3.2, define `OLD_KPATHSEA` additionally. For kpathsea older than 3.0, `VERY_OLD_KPATHSEA` must be also set. For MikTeX, define `HAVE_MIKTEX`. If none of these macros is defined, a simple *fopen*( ) will be used instead.

```
⟨ Include files 10 ⟩ +≡
#if defined (HAVE_LIBKPATHSEA)
#ifdef VERY_OLD_KPATHSEA
#include "kpathsea/proginit.h"
#include "kpathsea/progname.h"
#include "kpathsea/tex-glyph.h"
#else
#include "kpathsea/kpathsea.h"
#endif
#elif defined (HAVE_EMTEXDIR)
#include "emtexdir.h"
#elif defined (HAVE_MIKTEX)
#include "miktex.h"
#endif
```

70.

```
⟨ Global variables 2 ⟩ +≡
#if defined (HAVE_LIBKPATHSEA)
#ifdef KPSEDLL                                   /* this is kpathsea 3.3 and newer */
  extern KPSEDLL char *kpathsea_version_string;
#else
  extern DllImport char *kpathsea_version_string;
#endif
#elif defined (HAVE_EMTEXDIR)
  char emtex_version_string[] = "emTeXdir";
#elif !defined (HAVE_MIKTEX)
  char no_version_string[] = "no␣search␣library";
#endif
```

71.

```
⟨ Prototypes 11 ⟩ +≡
  char *TeX_search_version(void);
```

72.

```
  char *TeX_search_version(void)
  {
#if defined (HAVE_LIBKPATHSEA)
    return kpathsea_version_string;
#elif defined (HAVE_EMTEXDIR)
    return emtex_version_string;
#elif defined (HAVE_MIKTEX)
```

```
    char buf[200];

    strcpy(buf, "MiKTeX␣");
    miktex_get_miktex_version_string_ex(buf + 7, sizeof (buf) − 7);
    return buf;
#else
    return no_version_string;
#endif
  }
```

73.

```
⟨ Global variables 2 ⟩ +≡
#ifdef HAVE_EMTEXDIR
  struct emtex_dir cfg_path, hbf_path;
#endif
```

74.

```
⟨ Prototypes 11 ⟩ +≡
#ifdef HAVE_EMTEXDIR
  extern int setup_list(struct emtex_dir *, char *, const char *, unsigned);
  int dir_setup(struct emtex_dir *, const char *, const char *, unsigned);
  char *file_find(char *, struct emtex_dir *);
#endif
```

75.

We slightly modify *emtex_dir_setup*( ) (from the file `emtexdir.c`) to output a warning in case the environment variable *env* isn't set properly.

```
#ifdef HAVE_EMTEXDIR
  int dir_setup(ed, env, dir, flags)
      struct emtex_dir *ed;
      const char *env;
      const char *dir;
      unsigned flags;
   {const char *val;
    char path[260];

    ed→alloc = 0;
    ed→used = 0;
    ed→list = Λ;

    if (env ≠ Λ ∧ (val = getenv(env)) ≠ Λ)
      return setup_list(ed, path, val, flags);
    else
      fprintf(stderr, "Environment␣variable␣‘%s’␣not␣set;␣use␣current␣directory\n", env);

    return TRUE;
   }
```

76.

⟨ Global variables 2 ⟩ +≡
char *name_buffer*[FILE_NAME_LENGTH + 1];

77.

```
char *file_find(name, list)
    char *name;
    struct emtex_dir *list;
  {if (emtex_dir_find(name_buffer, sizeof (name_buffer), list, name, EDF_CWD))
    return name_buffer;
   return Λ;
  }
#endif
```

78.

For emtexdir we use the environment variables `HBFCFG` and `HBFONTS` for configuration resp. HBF header files.

⟨ Initialize TEX file searching 78 ⟩ ≡

```
#if defined (HAVE_LIBKPATHSEA)
#ifdef OLD_KPATHSEA
  kpse_set_progname(argv[0]);
#else
  kpse_set_program_name(argv[0], "hbf2gf");
#endif
#ifdef VERY_OLD_KPATHSEA
  kpse_init_prog("HBF2GF", 300, "cx", true, "cmr10");
#else
  kpse_init_prog("HBF2GF", 300, "cx", "cmr10");
#endif
#elif defined (HAVE_EMTEXDIR)
  if (!dir_setup(&cfg_path, "HBFCFG", Λ, EDS_BANG))
   {fprintf(stderr, "Couldn't␣setup␣search␣path␣for␣configuration␣files\n");
    exit(1);
   }
  if (!dir_setup(&hbf_path, "HBFONTS", Λ, EDS_BANG))
   {fprintf(stderr, "Couldn't␣setup␣search␣path␣for␣HBF␣header␣files\n");
    exit(1);
   }
#endif
```

This code is used in section 4.

79.

Finally, here are the searching routines. A special format in the kpathsea library for fonts which are neither PostScript nor TrueType (MISCFONTS) is available with version 3.3 and newer. For older versions we use the path for PostScript fonts (T1FONTS) to find HBF files. Configuration files are searched in the path specified within TEXCONFIG for old kpathsea versions, and within HBF2GFINPUTS for new versions.

```
⟨ Prototypes 11 ⟩ +≡
  char *TeX_search_cfg_file(char *);
  char *TeX_search_hbf_file(char *);
```

80.

```
#if defined (HAVE_LIBKPATHSEA)
  char *TeX_search_cfg_file(name)
      char *name;
   {
#ifdef OLD_KPATHSEA
     return kpse_find_file(name, kpse_dvips_config_format, TRUE);
#else
     return kpse_find_file(name, kpse_program_text_format, TRUE);
#endif
   }
  char *TeX_search_hbf_file(name)
      char *name;
   {
#ifdef VERY_OLD_KPATHSEA
     return kpse_find_file(name, kpse_dvips_header_format, TRUE);
#else
#ifndef KPSEDLL
     return kpse_find_file(name, kpse_type1_format, TRUE);
#else
     return kpse_find_file(name, kpse_miscfonts_format, TRUE);
#endif
#endif
   }
#elif defined (HAVE_EMTEXDIR)
  char *TeX_search_cfg_file(name)
      char *name;
   {return file_find(name, &cfg_path);
   }
  char *TeX_search_hbf_file(name)
      char *name;
   {return file_find(name, &hbf_path);
   }
#elif defined (HAVE_MIKTEX)
  char *TeX_search_cfg_file(name)
      char *name;
   {char result[_MAX_PATH];
```

```
    if (!miktex_find_input_file("hbf2gf", *name, result))
      return 0;
    return strdup(result);
   }
  char *TeX_search_hbf_file(name)
      char *name;
   {char result[_MAX_PATH];
    if (!miktex_find_miscfont_file(*name, result))
      return 0;
    return strdup(result);
   }
#else
  char *TeX_search_cfg_file(name)
      char *name;
   {return name;
   }
  char *TeX_search_hbf_file(name)
      char *name;
   {return name;
   }
#endif
```

81. An example.

This is the example configuration file `b5so12.cfg` (for use with DOS or OS/2 and the emtexdir searching engine):

```
hbf_header      et24.hbf
mag_x           2.076
x_offset        3
y_offset        -8
comment         fanti songti 24x24 pixel font scaled and adapted to 12 pt

design_size     12.0

nmb_fonts       -1

output_name     b5so12

dpi_x           300
checksum        123456789
coding          codingscheme Big 5 encoded TeX text

long_extension  no
job_extension   .cmd
rm_command      del
cp_command      copy
pk_directory    $HBF_TARGET\pk\360dpi\
tfm_directory   $HBF_TARGET\tfm\
```

If you say e.g.

```
set HBF_TARGET=c:\emtex\texfonts
```

on your DOS prompt (or in your `autoexec.bat` file), then the interpolated value of the `tfm_directory` keyword is `c:\emtex\texfonts\tfm\`. The HBF header file `et24.hbf` will be searched in the path specified by the `HBFCFG` environment variable.

The call

```
hbf2gf b5so12.cfg
```

creates the files

`b5so1201.gf`, `b5so1202.gf`, . . . , `b5so1255.gf`, `b5so12.pl`, and `b5so12.cmd`

After calling

```
b5so12.cmd
```

you will find the `PK` files in the `c:\emtex\texfonts\pk\360dpi` directory and the `TFM` files in the `c:\emtex\texfonts\tfm` directory; all `GF` files and `b5so12.pl` will be deleted.

The call

```
hbf2gf -n b5so1220 417
```

creates two files:

`b5so1220.gf` and `b5so1220.pl`

using the configuration file `b5so12.cfg`. The `GF` file would be named `b5so1220.417gf` if the flag `-n` had not been used.

It is possible to convert bitmap fonts to `PK` files almost automatically. The HBF header file already has the entry `HBF_BITMAP_BOUNDING_BOX` which defines vertical and horizontal offsets (in pixels), but these values are not in all cases optimal. If you omit `x_offset` and `y_offset` in the configuration file, the third and fourth parameter of `HBF_BITMAP_BOUNDING_BOX` is used, scaled to design size (to say it with other words: `x_offset` and `y_offset` will always apply to the design size to be synchronous with the `TFM` files).

Don't confuse scaling and magnification: Scaling here means that you choose a (arbitrary) design size and compute scaling values (`mag_x` and `mag_y`) which scales the bitmap to this particular design size at a certain (arbitrarily chosen) resolution (`dpi_x` and `dpi_y`). Magnification means that the scaled bitmap will be then magnified to a certain target size while still using the font parameters (i.e., the `TFM` file) of the design size.

In the sample, you have a 24×24 bitmap font which will be scaled to 12 pt having a resolution of 300 dpi:

1 pt are $300/72.27 = 4.1511$ pixel;

12 pt are $4.1511 * 12 = 49.813$ pixel;

thus the theoretical scaling value is $49.813/24 = 2.076$.

But especially for small sizes, this may not be the best value if the font should harmonize with, say, Knuth's Computer Modern fonts. I recommend to compute, say, 5 `PK` fonts, then check the CJK font with different TeX fonts to see whether the offsets and/or the scaling value is good. The greater the design size the finer you can control the offsets—as an example you could use a design size of 30 pt (nevertheless there is a compile-time constant `MAX_CHAR_SIZE` which limits the maximal character size; default is 255 pixels).

If you have found optimal offsets, you can produce many different magnifications of the CJK font using the same set of `TFM` files analogous to ordinary TeX fonts; as a simplification, we assume that `PK` files with a resolution of 300 dpi and a design size of 10 pt have the extension '`.300pk`' (respectively come into a '`300dpi`' subdirectory)—this is the reason why in the above example for the 12 pt design size a '`360dpi`' target directory has been used. Now we can use the following formula:

$$\mathit{needed\_dpi} = \mathit{your\_horizontal\_resolution} * \frac{\mathit{your\_target\_size}}{10.0}$$

Example: assuming that your printer has a resolution of 300×400 dpi, and you want 14.4 pt:

$$300 * \frac{14.4}{10.0} = 432$$

The vertical scaling value is $400/300 = 1.3333$. Use these values now to call `hbf2gf` in METAFONT-like mode:

```
hbf2gf b5so1220 432 1.3333
```

82. Index.

⟨ Check for environment variables 66 ⟩ Used in section 65.
⟨ Check other arguments 8 ⟩ Used in section 4.
⟨ Get code range 61 ⟩ Used in section 55.
⟨ Get sub code range 63 ⟩ Used in section 55.
⟨ Global variables 2, 15, 19, 27, 38, 49, 53, 58, 60, 62, 70, 73, 76 ⟩ Used in section 4.
⟨ Handle extension 56 ⟩ Used in section 55.
⟨ Include files 10, 69 ⟩ Used in section 4.
⟨ Initialize TEX file searching 78 ⟩ Used in section 4.
⟨ Initialize variables 28 ⟩ Used in section 4.
⟨ Necessary parameters 57 ⟩ Used in section 55.
⟨ Optional parameters 59 ⟩ Used in section 55.
⟨ Post section 23 ⟩ Used in section 21.
⟨ Postpost section 24 ⟩ Used in section 21.
⟨ Print help information 6 ⟩ Used in section 7.
⟨ Print version 5 ⟩ Used in section 7.
⟨ Process rest of line 44 ⟩ Used in section 42.
⟨ Produce a temporary row 33 ⟩ Used in section 32.
⟨ Prototypes 11, 13, 16, 20, 25, 29, 36, 39, 41, 45, 47, 50, 54, 64, 67, 71, 74, 79 ⟩ Used in section 4.
⟨ Scale X from *temp_pixelrow*[ ] into *new_pixelrow*[ ] and write it into *out_char*[ ] 34 ⟩ Used in section 31.
⟨ Scale Y from *pixelrow*[ ] into *temp_pixelrow*[ ] 32 ⟩ Used in section 31.
⟨ Scale row by row 31 ⟩ Used in section 30.
⟨ Scan options 7 ⟩ Used in section 4.
⟨ Search blank lines 43 ⟩ Used in section 42.
⟨ Special section 22 ⟩ Used in section 21.
⟨ Write character 18 ⟩ Used in section 17.
⟨ Write files 9 ⟩ Used in section 4.
⟨ Write out a row 35 ⟩ Used in section 34.

The `hbf2gf` program

(CJK Version 4.3.0)