New font tools for TEX

Werner Lemberg
Kleine Beurhausstr. 1
D-44137 Dortmund
Germany
a7971428@unet.univie.ac.at

Abstract

The following font tools will be described in this paper:

- VFlib, the *Vector Font Library*, which has been written by KAKUGAWA Hirotsugu (角川 裕次). This library together with an included suite of demo applications is aimed to provide a uniform interface for accessing fonts in many different formats. With the help of configuration files it is possible to access fonts with other encodings, to define new (virtual) fonts, and others. Using VFlib as a basis for `dvi` drivers, it will be no longer necessary to have `pk` files for any font format except for METAFONT.
- FreeType, developed by David TURNER, Robert WILHELM, and the author of this paper. This is a platform-independent library to render bitmaps from TrueType fonts. What makes it different from other freely available TrueType tools is a TrueType interpreter to process hints.
- `ttf2pk` and `hbf2gf`, maintained resp. written by the author. Both programs are part of the CJK package and are used to convert CJK TrueType and HBF fonts into TEX fonts.

Introduction

None of the tools are really new, but until now they either have not been presented to the (English) TEX community or have been hidden in other packages. Most of them are work in progress, and there is a good chance that this paper is already out of date when it is printed.

All of them have a CJK[1] background more or less; KAKUGAWA originally wanted easy Japanese font support for `dvi` drivers, my two programs are still only useful for CJK fonts, and FreeType was also soon extended to manage CJK TrueType fonts. Nevertheless, work is going on to internationalize the tools, making them useful for a broader audience.

In the bibliography you can find the locations from which to download the packages.

VFlib

Until recently documentation was only available in Japanese (see [2]) — the main reason why this great tool is virtually unknown outside of Japan. Now a translation into English has been made; this quote from `basic.txt` tells what VFlib is:[2]

Today, we have many font files and many different font file formats. When we need software to display or print characters which does not depend on a windowing system and/or an operating system, we must write interface routines for accessing font files in each application software again and again. To do this, programmers must have knowledge of font file formats; it will be a hard task for programmers if the number of font formats that an application software supports becomes large.

VFlib is a font library written in C providing several functions to obtain bitmaps of characters. VFlib hides the font format of font files and provides a unified API for all supported font formats. Thus, programmers for application software need not have knowledge on font file formats. Instead, any software using VFlib can support various font file formats immediately.

[...]

Currently, VFlib supports the following font file formats: PCF, BDF, HBF, TrueType, GF, PK, TFM, *ShotaiKurabu* (書体倶楽部,

[1] Chinese/Japanese/Korean

[2] In general, I'm following the documentation files very closely, omitting not so important or too technical details.

a vector font format for Japanese Kanji),[3] and JG (another vector font format for Japanese Kanji). Especially, VFlib can be used as a font module for drivers/previewers of `dvi` files (TEX/LATEX); part of this package is a sample `dvi` previewer for X Windows written with only 400 lines of C code.

VFlib stands for *Vector Font library* — earlier versions of the library have supported Japanese vector fonts only, hence the name.

Basic concepts. The VFlib module consists of two parts: the library itself which must be linked to the application program, and a font database file (called 'vflibcap' since the file format resembles the format of a termcap database; see below for an example).

Font classes and font drivers. VFlib can handle multiple font file formats. Reading a font file is done by an internal module in VFlib corresponding to the font file format. This internal module is called a 'font driver'. Service units provided by font drivers are called 'font classes'. From an end-user's point of view, font formats are distinguished by the names of font classes. Font drivers themself are not visible for end-users.

Some font drivers may not read font files on disk; they may generate glyphs and outlines by internal computation only. In addition, some font drivers may return glyphs which are obtained as glyphs by another font class.

Font names and searching. In VFlib, a font is specified by a 'font name' on opening. First, VFlib checks if the font name is given in vflibcap or not. If the font name is found, VFlib reads the description for the font in vflibcap, invokes a font driver corresponding to the font class name and opens the font file.

If the font name is not given in a vflibcap file, a font searching mechanism is invoked. Since there are so many font files for X Window and TEX, this feature has been introduced to avoid writing an entry for each font file. Various font drivers will be called to see whether the font can be opened; a list of font drivers for font searching is given in the vflibcap file.

Fonts described in a vflibcap file are called 'explicit fonts' and fonts that are searched by the font search feature are called 'implicit fonts'.

For TEX fonts, the kpathsea library will be used for searching.

The vflibcap database. Each (virtual) font as provided by VFlib has its inherent information on point size, pixel size, and resolution of the target device. In addition to these font metrics are defined for each glyph.

Some font file formats do not have such concepts; in this case, missing information either is given in a vflibcap file or the specific font driver provides default values. For instance, a TrueType font is a vector font and is not restricted to a certain point size and resolution of the target device (since vector fonts can be scaled to any size). Another example is the ShotaiKurabu font format which does not have font metric information at all: a font driver for this font format generates virtual font metrics using the data given in a vflibcap file.

Here a small excerpt of a vflibcap file suitable for Japanese TEX (JTEX); omissions are indicated with three dots.

```
VFlib-Defaults:\
  :implicit-font-classes=ascii-jtex-kanji,\
                         gf/pk:\
  :extension-hints=pk=ascii-jtex-kanji,\
                   gf=ascii-jtex-kanji,\
                   pk=gf/pk,\
                   gf=gf/pk,\
                   .ttf=ttf:\
  :variables-default-values=\
    $TeX_KPATHSEA_PROGRAM=\
      /usr/local/teTeX/bin/xldvi:

TeX-Defaults:\
  :kpathsea-mode=ljfour:\
  :dpi=600:\
  :kpathsea-program-name=\
    $TeX_KPATHSEA_PROGRAM:

TrueType-Defaults:\
  :extension=.ttf:\
  :aspect=1:\
  :dpi=600:\
  :font-directories=\
    /dos/texmf/fonts/truetype/japanese:\
  :platform=microsoft:


mincho-jtex:\
  :font-class=ttf:\
  :font-file=uwjmg3.ttf:\
  :magnification=0.92:\
  :writing-direction=h:\
  :character-set=jisx0208_1983:\
  :encoding-force=sjis:

mincho-5pt:\
  :point-size=5:\
  :inheritance=mincho-jtex:

...
```

[3] Other transcription forms are *SyotaiKurabu* or *Syotai-Club*.

```
mincho-10pt:\
  :point-size=10:\
  :inheritance=mincho-jtex:


min:\
  :font-class=ascii-jtex-kanji:\
  :kanji-adjustment-file=\
    /usr/local/VFlib/ascii-jtex/ttf.adj:

min5:\
  :kanji-font=mincho-5pt:\
  :inheritance=min:

...

min10:\
  :kanji-font=mincho-10pt:\
  :inheritance=min:
```

The format of a capability file is somewhat strange: each entry must be formally on one line (which can be split with a trailing backslash). The name of a capability entry is the first name starting a line; all capability descriptions then follow in the format

:⟨*entry 1*⟩:⟨*entry 2*⟩:...:

A single colon ':' is equivalent to the construction ':⟨*whitespace*⟩:', making it convenient to break a line after the colon. Capability descriptions have the format

⟨*cap description*⟩=⟨*cap value*⟩

The first equal sign separates the description from the value (which can be e.g. a list containing equal signs too).

The capability entry 'VFlib-Defaults' defines global default values for VFlib. 'implicit-font-classes' specifies a list of font classes for implicit font search; the font class drivers are invoked in the order of that list for searching. 'extension-hints' gives an ordered list of pairs indicating which extension of an implicit font needs which driver to handle. In the above example there are e.g. two entries for files ending with pk: one for Japanese TEX and one for standard TEX. If the first driver fails, the second will be called. Finally, 'variables-default-values' gives a list of default values which can be overridden at run-time if supplied as an argument string to the initialization function of VFlib.

'TeX-Defaults' primarily gives initialization values for kpathsea. Users of web2c and teTEX should be quite familiar with the description names and its meanings.

All other entries are used to define a Japanese TrueType font as a font for JTEX. The final JTEX font is built from various layers, starting with the entry 'mincho-jtex' which defines the TrueType font name, the writing direction, the character set, etc. 'mincho-5pt' is an example of how to inherit font capabilities: only a point size declaration has been added. Then the 'min' base font class is specified, calling the 'ascii-jtex-kanji' font driver and using an adjustment file `ttf.adj` for all fonts of this class.[4] Note that in 'min' no fonts are defined — the concept is similar to object oriented languages where some base classes are defined on which virtual classes are built. With 'min5' or 'min10' the top level is reached, using all previously constructed font classes.

NFSS would not require size quantization of the font; it's easy to add a proper entry like this:

```
min-nfss:\
  :kanji-font=mincho-jtex:\
  :inheritance=min:
```

A font defined in this way can then be used at any size; VFlib would scale the font appropriately.

The VFlib API. The number of functions is small due to the identical interface for all font formats. Before opening any font you have to call `VF_Init` to initialize the library with a vflibcap file. Opening and closing of a font are handled with the functions `VF_OpenFont` and `VF_CloseFont` respectively. A glyph bitmap can be accessed in two ways. Either you specify bitmap sizes in pixels (`VF_GetBitmap2`), or you pass the resolution (in dpi) together with the point size as parameters (`VF_GetBitmap1`). Similar commands exist for getting outline (vector) data (`VF_GetOutline`) and for obtaining information on metrics (`VF_GetMetric1`, `VF_GetMetric2`). All the functions take the character code as a mandatory parameter.

Auxiliary functions are provided to free bitmap or metrics objects, to copy or scale bitmaps, and to 'dump' the glyph using ASCII characters to get similar output as `gftype`.

Finally, you can write your own font driver (to be installed with `VF_InstallFontDriver`). Font drivers must provide a small set of glyph manipulating functions,[5] and pointers to those functions are then passed to the VFlib engine. Using the function `VF_GetProp` it's easy to extract font-class-specific

[4] This file compensates the mono-width of Japanese glyphs with small offsets for certain character classes like CJK punctuation characters to improve typographical output. Character classes are a special feature of JTEX.

[5] With 'glyph' an empty box can be meant also e.g. for writing a `tfm` driver.

entries from the vflibcap file to control the new font driver.

Limitations and planned features. Many parts of the VFlib package are still undocumented, or documentation is only available in Japanese. For instance, VFlib contains a complete library for interpreting `dvi` files (with specials) which further simplifies the writing of `dvi` drivers. A suite of `dvi` previewers and `dvi` drivers is also included.

Currently, VFlib is limited to UNIX-like operating systems, but it should not be difficult to port it to other platforms because no real dependencies on UNIX features are built in.

Another useful tool yet to be written is a module or program for creating `tfm` files from the bitmap and vector fonts.

Planned for the near future are modules for processing PostScript fonts and TEX virtual fonts; support for Ω metrics files is already implemented, but without a `vf` module its use is rather academic.

FreeType

The report on FreeType (see [6]) will cover only the basics without going too much into detail — the final end-user API has not been defined yet. It is not really linked to TEX, and if you are not interested in how a rasterizer works, you should skip this section. Nevertheless, it is linked to typography, and the text presents some general principles of how outline glyphs will be handled to yield bitmaps.

FreeType is developed in a rather unusual way. The package provides the complete library code and some tools which demonstrate the use of the library in two programming languages, namely in C and in PASCAL. We try to keep the library small (about 60 kByte if compiled for maximal speed with `gcc`); nevertheless it is highly portable since the C part is written in ANSI C, having only a few architecture dependencies which can be adjusted with a few global macro definitions.

The rasterizer. A rasterizer converts the vector data of a glyph into a pixel representation. In this short overview all complications (drop-out control, wrong contour direction, sub-banding of profiles etc.) are omitted.[6] This part of the FreeType engine is quite generic and could be adapted to, say, PostScript fonts too.

Glyphs as stored in a TrueType font (see [5] for a reference) consist of vectorial information (straight lines and Bézier curves of second order[7]) together with hinting instructions which move the points determining the glyph contours to device resolution dependent locations before rasterization.

We now assume that all point moving has been done, and that the x and y coordinates of the points are stored in a list together with a flag to indicate whether the point is *on* or *off* the curve. See Figure 1.[8]

A *scanline* is a pixel line in the target bitmap. An *outline*, also called *contour*, is a closed line that delimits an inner and an outer region of the glyph. The best way to fill a shape is to decompose it into simple horizontal segments, called *spans*. Spans are computed for each scanline. This is usually done from the top to the bottom of the shape, in a movement called *sweep* (see Figure 2). It's easy to see that there is typically more than one span per scanline. For each scanline during the sweep operation we need the horizontal (x) coordinates of the start and end points of all spans. These are computed before the sweep, in a phase called 'decomposition' which converts the glyph contours into *profiles*.

Profiles are sections of the contours which are either only ascending or only descending, i.e. *monotonic* in the vertical direction (we will also say y-monotonic). It can easily be deduced from Figure 3 that it is possible to resolve any contour into vertical profiles and horizontal lines (which are *not* part of a profile).

Each profile inherits the direction of the parent contour (this is necessary to decide whether a point is inside or outside of a contour, see below). Figure 4 shows that a contour can have multiple profiles. Profiles are also called 'edges' or 'edgelists' in other graphics libraries.

The rasterizer stores a profile as an array of x coordinates of the intersection points of the profile and the affected scanlines. To allocate a profile array without wasting memory we must know the height of that profile; with other words, we have to compute the vertical extrema (minimum and maximum). This can be done very easily for straight lines, but it is not trivial for Bézier arcs because

[6] The original document is `raster.doc` of the FreeType package, written by David TURNER.

[7] PostScript fonts use third-order Bézier curves.

[8] A second-order Bézier curve (also called quadratic spline) is fully specified with the starting point of the curve, a control point usually off the line, and the end point of the curve. It is possible to have two consecutive off-points in the points list; in this case a virtual on-point between the two off-points will be constructed. The parametric form of a quadratic spline is

$$p(t) = (1-t)^2 p_1 + 2t(1-t)p_2 + t^2 p_3 \quad ;$$

t denotes a real number in the range $[0,1]$, p_1 is the start point, p_2 the control point, and p_3 the end point.

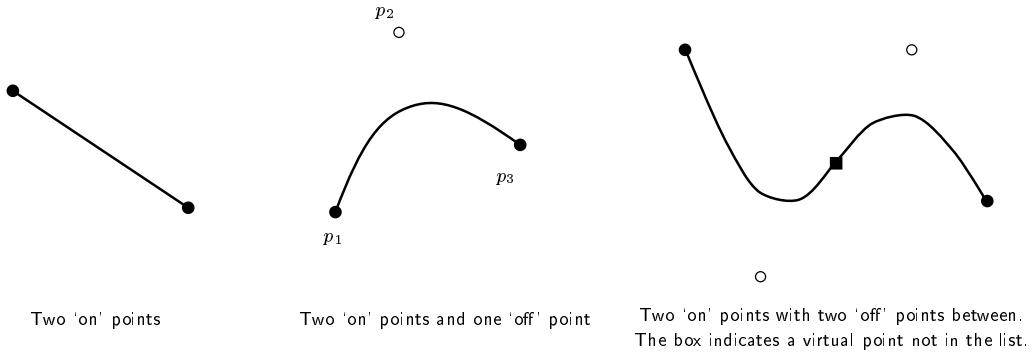

Figure 1: Possible 'on' and 'off' point combinations in a FreeType font

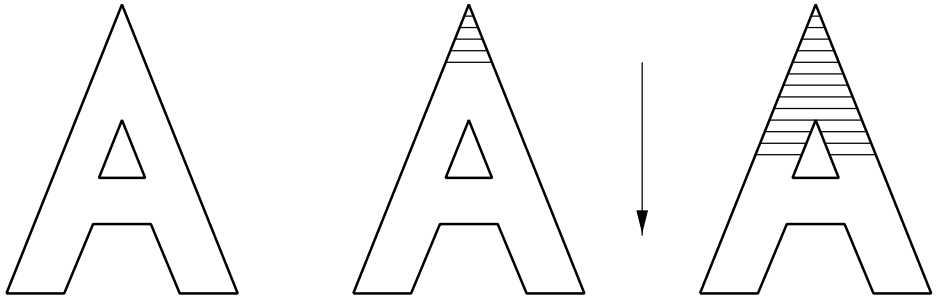

Figure 2: Filling a shape with spans. The arrow indicates the sweeping direction.

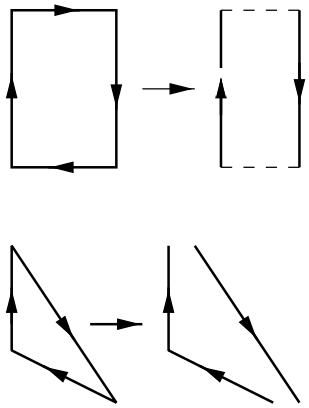

Figure 3: Decomposition of a contour into profiles

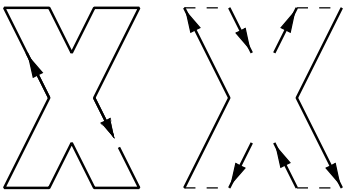

Figure 4: A contour with multiple profiles

they are not monotonic in the general case.[9] Nevertheless, a Bézier arc can be split into two subarcs with very little computation. Both subarcs are again Bézier arcs, and one of them is guaranteed to be y-monotonic. Look at Figure 5: p_i denote the points belonging to the original curve, q_i and r_i then define the subarcs (i being 1, 2, or 3). The following formulæ give the relationship between an arc and its subarcs:

$$q_1 = p_1; \quad q_2 = (p_1 + p_2)/2$$
$$r_3 = p_3; \quad r_2 = (p_2 + p_3)/2$$
$$q_3 = r_1 = (q_2 + r_2)/2$$

We stop if either all subarcs are monotonic or the subarcs become too small; in both cases we've found an extremum. This process is called *flattening*.

The next step is to compute all intersection points of profiles and scanlines. In the case of lines this is straightforward, but it is a little more complicated for splines. Fortunately we can use arc splitting again. Consider Figure 6. The horizontal lines represent scanlines, and a short segment of a profile is shown. If we continue splitting until each subarc

[9] A quadratic spline is y-monotonic if and only if the y coordinates of the points p_1, p_2, and p_3 are monotonic, i.e. $p_{1_y} \le p_{2_y} \le p_{3_y}$ or $p_{1_y} \ge p_{2_y} \ge p_{3_y}$. If $p_{1_y} = p_{2_y} = p_{3_y}$, the arc degenerates into a horizontal line.

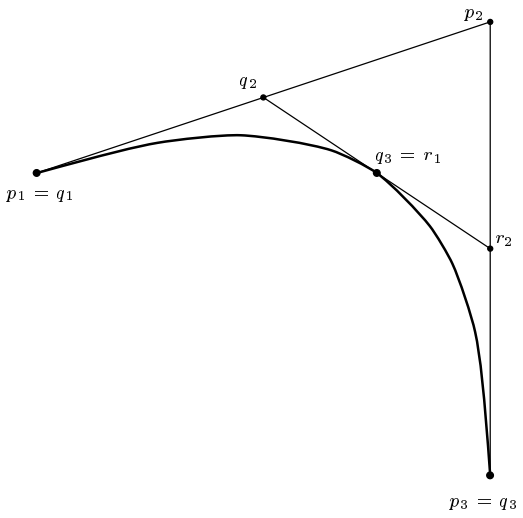

Figure 5: splitting a Bézier arc into two subarcs

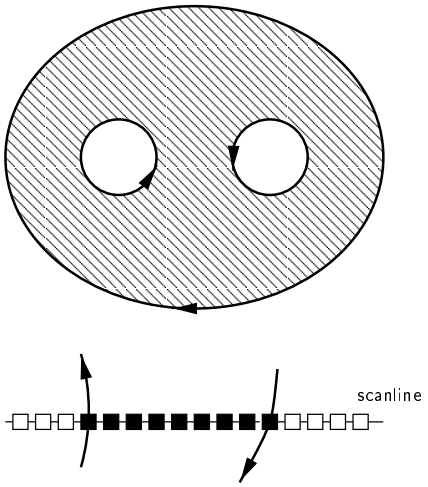

Figure 7: the interior of a glyph must always be to the right of the contour

crosses only one scanline, we can safely approximate the subarcs with straight lines for which the computational effort is minimal. Internally this has been realized with a Bézier arc stack; if the topmost arc can be replaced with a line, the intersection point is computed and the arc is popped off the stack. Otherwise the arc is popped off, then split, and the two new subarcs are pushed on the stack. This will be repeated until the stack is empty.

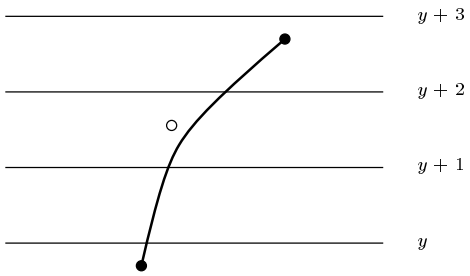

Figure 6: the stepping process: replacing sufficiently small subarcs with lines

We are done. The contours have been resolved into profiles, and the profiles have been decomposed into intersection points of the profiles and the scanlines. One last thing must be taken into account: how can we decide which side of the countour is interior and which is exterior? The TrueType specification defines that the interior is always on the right side of the contour (see Figure 7). Having an intersection point together with the contour direction, we can decide simply which pixels must be blackened.

The instruction interpreter. In figure 8 you can see what instructions basically do. To avoid dropouts or ugly shapes, points are moved to new (resolution dependent) locations before rendering, assuring that even for low resolutions good optical results can be computed. It has turned out that the TrueType specifications are often very fuzzy about certain instructions. Long debugging sessions with well hinted TrueType fonts were needed, comparing the results with rendered bitmaps of other (commercial) TrueType engines, to find out the undocumented behaviour of those instructions. Nevertheless, after mastering these obstacles, most glyphs are now rendered equally well with FreeType as with the rasterizers of Windows and the Mac.

To be added in the near future is instruction support for composite glyphs. Again the specifications are too fuzzy to allow a straightforward implementation without testing undocumented instruction properties — for instance, should the instruction code of the subglyphs be executed or only the instructions for the composite glyph? The TrueType specification says nothing about this problem.

Font tools from CJK

Both programs discussed in this section are part of the CJK package ([3]). They are specialized to CJK fonts, but work is going on to internationalize them. Script file skeletons of `MakeTeXPK` et al. are delivered with these utilities for on-the-fly font generation.

`ttf2pk`. It is currently a special tool for converting CJK TrueType fonts into `tfm` and `pk` files, but later

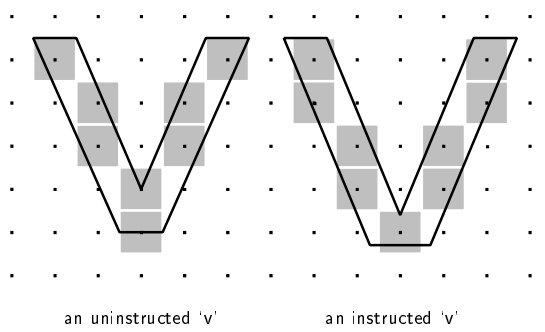

Figure 8: comparison between a hinted and an unhinted glyph

on it will use FreeType to process all kinds of TrueType fonts. The original author is Lin Yaw-Jen who modeled `ttf2pk` after `ttf2bmp` and `pbmtopk`.

`ttf2pk` takes a non-composite CJK TrueType font in Big 5, EUC,[10] or SJIS encoding and converts a certain contiguous subrange (at most 256 characters) into a `pk` and a `tfm` file. The program has a lot of command switches; a typical call looks like

```
ttf2pk ntukai01.pk ntukai01.tfm 600 1.0 \
       0xA140 256 -e Big5 ntu_kai.ttf
```

'`ntukai01`' is the first subfont of the ntukai font, '`600`' is the resolution in dpi, '`1.0`' a vertical scaling factor (for printers with different horizontal and vertical resolutions), '`0xA140`' the first character code of the subfont, and '`256`' the number of characters in the subfont. The switch `-e` selects the encoding of the TrueType font, and the last parameter is a full path to the TrueType font file.

Most of the other options not shown here have been inherited from `pnmtopk`; one parameter ('`-r`') has been added to rotate the glyphs by 90 degrees, enabling faked vertical typesetting with TEX.[11]

CJK subfonts as used in the CJK package are discussed in another paper of the proceedings ([4]).

`hbf2gf`. Similarly to `ttf2pk`, `hbf2gf` is used to split CJK bitmap fonts into subfonts. Its source code is written in CWEB; the format used by this tool is the *Hanzi Bitmap Font* format (HBF, see [1] for a complete reference). Basically the format consists of the bitmap files and a header file describing the font. Here an example of an HBF header file, describing a Chinese font with the character set CNS plane 7:

```
HBF_START_FONT 1.1
HBF_CODE_SCHEME CNS11643-92p5
FONT cns40st-5
SIZE 40 150 150
HBF_BITMAP_BOUNDING_BOX 40 40 0 -6
FONTBOUNDINGBOX 40 40 0 -6
STARTPROPERTIES 23
FONTNAME_REGISTRY ""
FOUNDRY "CBS"
FAMILY_NAME "Song"
WEIGHT_NAME "medium"
SLANT "r"
SETWIDTH_NAME "normal"
ADD_STYLE_NAME "fantizi"
PIXEL_SIZE 40
POINT_SIZE 400
RESOLUTION_X 75
RESOLUTION_Y 75
SPACING "c"
AVERAGE_WIDTH 400
CHARSET_REGISTRY "CNS11643.92p5"
CHARSET_ENCODING "0"
WEIGHT 19329
RESOLUTION 110
X_HEIGHT 34
QUAD_WIDTH 40
FONT_ASCENT 34
FONT_DESCENT 6
DEFAULT_CHAR 0x2121
ENDPROPERTIES
COMMENT "This HBF header file is in the"
COMMENT "public domain."
HBF_START_BYTE_2_RANGES 1
HBF_BYTE_2_RANGE 0x21-0x7E
HBF_END_BYTE_2_RANGES
HBF_START_CODE_RANGES 1
HBF_CODE_RANGE 0x2121-0x7C51 4040w5.bin 0
HBF_END_CODE_RANGES
COMMENT
COMMENT Rarely used characters defined by
COMMENT Ministry of Education of Taiwan,
COMMENT said to be disjoint from the
COMMENT previous planes.
COMMENT 8603 characters, 2121--7C51.
COMMENT
HBF_END_FONT
```

The syntax is very similar to the format of a BDF header (bitmap fonts used with X Windows); a small set of keywords (starting with '`HBF_`'; all others are BDF specific) have been added to accommodate the special needs of CJK files. As an example, the line

```
HBF_CODE_RANGE 0x2121-0x7C51 4040w5.bin 0
```

says that the file `4040w5.bin` contains glyphs with the character codes `0x2121`–`0x7C51`, starting at offset 0. '`HBF_BYTE_2_RANGE`' specifies the valid range of the second bytes of the double byte font encoding

[10] EUC stands for Extended UNIX code; examples are Chinese GB, Japanese JIS, Korean KS encoding.

[11] Alas, only Big 5 encoding has both horizontal and vertical punctuation marks, but even here the set is not complete. For typographically satisfying results you need a font intended for vertical typesetting.

(see [3] for more details). Both keywords can appear more than once.

`hbf2gf` can be called in two 'modes': the first creates a complete set of subfonts for a particular HBF font, and the second computes the `gf` and `tfm` file of one subfont (to be used in `MakeTeX...` scripts). In both cases, a configuration file is used to avoid lengthy parameter lists.

We continue the example from above with an `hbf2gf` configuration file using the just defined CNS font (long lines are splitted and marked with a final backslash for clarification):

```
hbf_header \
 $TEXMF/fonts/hbf/chinese/cns40/cns40-5.hbf

comment \
 CNS plane 5 song 40x40 pixel font \
 scaled and adapted to 12pt

mag_x          1
design_size    12.0

x_offset       2
y_offset       -8

output_name    c5so12

checksum       123456789

dpi_x          300

pk_files       no
tfm_files      yes

pk_directory \
 $TEXMF/fonts/pk/modeless/chinese/c5so12/
tfm_directory \
 $TEXMF/fonts/tfm/chinese/c5so12/
```

Keywords must start a line; a line not starting with a known keyword is ignored. Environment variables can be specified with a starting dollar sign.

An important concept to understand `hbf2gf` configuration files is the difference between 'magnification' and 'scaling'.[12] The former denotes a scaling factor to reach the design size of the font (in the above example it is 1.0 to get 12 pt). Offset values (given in pixels) refer to this size. The latter then scales the font to its final size, indicated with `dpi_x` (and optionally `dpi_y`).

It is also possible to create fonts with slanted and rotated glyphs; additionally the next version will be able to produce Ω virtual fonts.

[12] The wording is a bit unfortunate because the meaning is different in TEX.

Conclusion

In a not too distant future the utilities and libraries described in this paper will more or less merge, providing a vital basis for handling fonts under Ω and TEX. With VFlib as the framework it will be possible to access already existing fonts in Unicode encoding regardless of the original encoding, create virtual fonts on the fly as needed for text processing with mixed writing directions (both horizontal and vertical), sophisticated space handling between fonts and much more.

With FreeType a new font world will be opened to TEX users working on UNIX like operating systems — after the integration of FreeType `ttf2pk` will be (hopefully) as useful as `pstopk` or `gsftopk`.

References

[1] Nelson Chin et al. Hanzi Bitmap Font (HBF) file format version 1.1. Available electronically from `ftp://ftp.ifcss.org/pub/software/info/HBF-1.1.tar.gz`, September 1994.

[2] Kakugawa Hirotsugu (裕次角川). The VFlib package. Available from `ftp://gull.se.hiroshima-u.ac.jp/1997`.

[3] Werner Lemberg. The CJK package. Available from CTAN, `language/chinese`, 1997.

[4] Werner Lemberg. The CJK package for LATEX 2ε — multilingual support beyond babel. In *Proceedings of TUG 97*, July 1997.

[5] Microsoft corporation. TrueType 1.0 font files. Available electronically from `ftp://ftp.microsoft.com/developr/drg/TrueType/ttspec.zip`, November 1995.

[6] David Turner, Robert Wilhelm, and Werner Lemberg. The FreeType package. Available from `ftp://ftp.physiol.med.tu-muenchen.de/pub/freetype`, 1997.