SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **30**:1–8 *Prepared using* speauth.cls *[Version: 2000/03/06 v2.1]*

A General Purpose Font Module for Multilingual Application Programs

H. Kakugawa*[1], M. Nishikimi[2], N. Takahashi[2], S. Tomura[2], and K. Handa[2]

[1] *Graduate School of Engineering, Hiroshima University, 1-4-1 Kagamiyama, Higashi-Hiroshima, Hiroshima, 739-8527 JAPAN (email: h.kakugawa@computer.org)*
[2] *National Institute of Advanced Industrial Science and Technology (AIST) 1-1-1 Umezono, Tsukuba, Ibaraki, 305-8568 JAPAN (email: {nisikimi,ntakahas,tomura,handa}@m17n.org)*

SUMMARY

Multilingual information processing technology has enabled us to handle the text of many languages and scripts. Technical difficulties, however, still remain in the field of displaying and printing text. Window systems and printers often lack support for scripts that are not widely used and there has been no simple and flexible remedy for this problem.

In this paper, we describe the general font module VFlib for multilingual text. VFlib knows how fonts are represented in different font formats and obtains glyphs as bitmap images from various font files. As VFlib provides a unified interface for font access, application programs can employ many font types without handling their details.

We have also implemented previewers and printer drivers that use VFlib as the font retrieval module. These application programs include a program named ps-multi that prints multi-script text created with the multilingual text editor Emacs Version 20 or later. From our experiences of implementing these application programs, we believe that VFlib is a simple and powerful solution for multi-script text printing and displaying.

KEY WORDS: VFlib; computer typography; fonts; multilingual information processing; Mule; GNU Emacs

*Correspondence to: Hirotsugu Kakugawa, Information Engineering, Graduate School of Engineering, Hiroshima University, 1-4-1 Kagamiyama, Higashi-Hiroshima, Hiroshima, 739-8527 JAPAN +81-824-24-7681

Received ???
Revised ???
Accepted ???

INTRODUCTION

Today people exchange e-mail and browse World Wide Web pages over the Internet. Since most application programs and operating systems were originally designed to use the English language, English is the de facto standard over the Internet now. However, most of the world's people need to read, write, or print documents in languages other than English. Research on multilingual, or more precisely, multi-script information technology aims to satisfy their requirements.

When we wish to print or display multi-script text, we often find that the window systems, printers, or application programs we use do not support the data formats of fonts we have at hand. For example, modern window systems often support fonts in the Type 1 and TrueType (or OpenType) formats to produce high quality glyph images using their hinting feature. However, we can print/display a multi-script text with these high quality glyph images only when all necessary fonts are available in the Type 1 or TrueType format. If this is not the case, we have to give up on printing/displaying the entire text. When high quality outline fonts for some scripts are unavailable, it is much better to have a low quality output with bitmap fonts than to have no output at all. We need a system that makes the best of what we have.

Some people consider that adopting Unicode [6] compliant systems and Unicode-based outline fonts would solve all these font problems, but this is not true, for the following reasons:

1. Many characters are still missing from Unicode, especially ancient ones; for example, neither Hieroglyphic scripts (ancient Egyptian) nor Glagolitic (old Slavic) are included. Concerning modern scripts, many Chinese, Japanese, and Korean (CJK) ideographic characters are known to be still missing. Without using system and user dependent private areas, there is no way to represent these missing characters with Unicode.
2. Although modern operating systems (Windows 2000, Solaris, etc.) have adopted Unicode to represent character data and a number of Unicode-based outline fonts are provided for Latin-based character sets, high quality fonts are not always available for other scripts.

To summarize, only a small number of scripts and languages benefit from Unicode compliant systems with current technology. Thus, we need a better mechanism to print and display a wide variety of characters.

In this paper, we present the font system library VFlib, which we have been developing to solve the problems mentioned above. VFlib knows how fonts are represented in different font formats and obtains glyphs as bitmap images from various font files. Since VFlib provides a unified interface for font access, application programs can employ many fonts without handling their details.

We first review and categorize a variety of font file formats proposed so far and explain why and in what circumstances a font module like VFlib is useful. Then we describe the font module VFlib itself and its use by application programs.

Prepared using speauth.cls

Fonts, font file formats, and their taxonomy

Many font file formats have been proposed so far and more may be proposed in the future. We believe that a font module should support as many font file formats as possible. However, different font file formats provide different kinds of glyph information (for example, some may provide ligatures but others may not) in different styles. If a font module can absorb such differences of information and provide a unified interface for handling fonts to application programs, developers of applications programs will be able to use various font files without being aware of the details of each font file format.

In order to design and implement a unified interface for different font file formats, we use a taxonomy, or categorization, of font file formats with respect to the following seven aspects:

1. *Glyph Description:* (a) bitmap, (b) outline, (c) stroke, or (d) no description.
   Descriptions using outlines or strokes utilize several methods to represent curves. The methods include Bezier, B-spline, and approximation by lines. Some font formats are defined to obtain glyphs from other fonts and have no glyph description of their own.
2. *Target Device Resolution Dependency:* (a) dependent on the resolution of the target device, or (b) independent.
   Outline fonts are usually independent of the target device resolution. However, special handling (called "hinting") may be used when rendering small glyphs on a low resolution device to get better quality.
3. *Dimension:* (a) pixel, (b) physical unit (point, etc.), or (c) logical unit.
   Dimension often relates to *Target Device Resolution Dependency*. Some bitmap font formats that do not depend on device resolution use pixels for dimension; others use points. Most outline font formats define their own logical units.
4. *Font Size:* (a) pixel, (b) physical unit (point, etc.), or (c) not specified.
   Most outline font formats have no information on font size since glyphs can be rendered in any size.
5. *Font Metrics:* (a) separate metrics for each glyph, (b) single metric for all glyphs, or (c) not given.
6. *Reference Point Movement:* (a) either horizontal or vertical movement, (b) both horizontal and vertical movements, (c) information that can represent any writing direction (diagonal, for example), or (d) not given.
   Reference Point Movement defines the writing direction and often relates to the type of *Font Metrics*. When a font format does not define *Reference Point Movement*, the application program is responsible for glyph arrangement. For instance, an application program might assume that glyphs are drawn from left to right and the reference point moves forward by a fixed distance. Many character terminals and terminal emulators adopt such a scheme because most of them use only one fixed-width bitmap font.
7. *Information on Kerning and Ligatures:* (a) contained in the font files or (b) not given.
   Typesetting programs need to know how characters in one font file should be kerned or ligated. Some font files come with that information. Other font files that do not specify kerning or ligatures usually contain glyphs for ligated characters. Typesetting programs

may look up other files in order to decide when and where those kerned or ligated glyphs should be used.

The latest version of VFlib, version 3.6, supports the font file formats listed below. Those formats cover an extensive range of currently available fonts. As VFlib provides a mechanism to extend itself, new font file formats can be supported easily if necessary.

- BDF [2]
  BDF (Glyph Bitmap Distribution Format) is a bitmap font format to distribute font files for the X Window System. Although outline CJK (Chinese, Japanese, and Korean) fonts may be difficult to obtain outside Asia, many BDF CJK fonts are widely and freely available.
- PCF [40]
  The PCF (Portable Compiled Font) format is another bitmap font format for the X Window System. A PCF font file is generated from a BDF font file using the converter program bdftopcf. The X Window System display server (hereafter "X server" for short) uses font files in this format to display glyphs.
- Syotai-Kurabu and JG
  These two formats, developed by a Japanese company, are outline font formats for the Japanese JIS X 0208 character set. Commercial fonts in these formats are no longer available today. There are several freely available font files for Japanese JIS X 0208 character set in Syotai-Kurabu format [49, 48].
- HBF [42]
  The HBF (Hanzi Bitmap Font) format is a bitmap font format for CJK characters. Some fonts are freely available in this format.
- TrueType [5, 10]
  The TrueType format is an outline font format developed by Apple and Microsoft. The TrueType specification supports hinting to produce better quality rasterization. To rasterize TrueType fonts, VFlib internally uses FreeType [47]. As FreeType supports hinting, it generates high quality bitmaps.
- Type 1 [1]
  The Type 1 format is an outline font format developed by Adobe Systems Inc. The Type 1 font specification also supports hinting. To rasterize Type 1 fonts, VFlib internally uses T1lib [36], which also supports the hinting feature.
- GF [28, 29, 44]
  The GF (Generic Font) format is a bitmap font format used by printer drivers and previewers of the TEX typesetting system [30, 31]. A font file in this format is generated by the METAFONT program [28, 29]. Input to METAFONT is an outline description of a font written in its own language. Together with outline font and target device information (e.g. device resolution), METAFONT generates a bitmap font file in the GF format and a font metric file in the TFM format (see below). GF font files are generated for each output device type and for some scaled sizes when the TEX system is installed.
- PK [44]
  The PK (Packed) font format is a bitmap font format for the TEX typesetting system. A PK font file is obtained by compressing a GF font file using the GFtoPK program [44].

Thus, a font in this format contains essentially the same information as its source PK font.

- Virtual Font [32]
  The virtual font format is used in the TEX system to rearrange or synthesize glyphs extracted from one or more existing fonts. (A font referred to by a virtual font is called a *base font* or *map font*.) A virtual font file defines each glyph via a small program that specifies which glyphs in which base fonts are used and how they are arranged or synthesized. Like GF or PK fonts, each virtual font has a corresponding TFM file. Since the TEX program itself uses only TFM files when typesetting documents, it does not care about the format of base fonts specified in virtual fonts and so the TEX program does not require any source code modifications to be used with VFlib. It is the responsibility of the previewer or print driver to produce images using information stored in a virtual font. This means that if the previewer or printer driver uses VFlib, base fonts referred to by virtual fonts can be in any format supported by VFlib.
- TFM [31]
  A TFM (TEX Font Metric) file is a font metric file for TEX and contains no glyph images. For every GF, PK, or Virtual Font file, there is a corresponding TFM file. VFlib can interpret a TFM file as a font file containing black (or white) rectangles of various sizes as glyph images. This is useful when some fonts in GF, PK, or Virtual Font formats are unavailable for some reason because characters in non-existing fonts can still be printed as black rectangles. Without this feature, printing would be aborted and no output is obtained at all.
- JFM [8]
  The JFM (Japanese Font Metric) format is an extension of TFM to handle Japanese Kanji characters for a variation of TEX localized for Japanese.
- OFM level 0 [41]
  OFM (Omega Font Metric) is an extension of TFM for the multilingual TEX Ω. It is used when Ω typesets a document.

Table I is a sample classification of the TrueType, BDF, and PK font formats by the taxonomy discussed above. Although TrueType does not contain ligature information, OpenType, which is an extension of TrueType, can contain ligature information [7, 9].

FONT MODULE

When we need to print or display text with a printer or a window system and the text contains a script that requires a font whose format is not supported by the printer or the window system, we have to develop one of the following:

1. a font file in a format that is supported by the printer or the window system, or
2. a program that has a rasterizer for the font file format.

With the former strategy, we can obtain high quality output for the printer or the window system in use. One drawback is that we have to create a font file for each script and typeface,

Prepared using speauth.cls

Table I. A sample classification of font formats in our taxonomy

font format	TrueType	BDF	PK
glyph description	outline	bitmap	bitmap
resolution dependency	independent	dependent	dependent
dimension	logical	pixel	physical
font size	unspecified	pixel&physical	physical
font metrics	each	each	each
reference point movement	horiz.&vert.	horiz.&vert.	horizontal
kerning/ligature	kerning	neither	neither

which uses a lot of computing resources. That is why we have chosen the latter approach and developed the font module (library) VFlib that can be used with many application programs.

The first version of VFlib, VFlib version 1, was designed to solve the problems of Japanese font formats. Around 1990, the authors had the TEX and LATEX programs localized for Japanese, a previewer on the X Window System, and a printer driver for laser printers installed on Sony NEWS workstations. Sony NEWS workstations had only two Japanese outline fonts but many Japanese outline fonts for PCs were commercially available and inexpensive even in those days. The problem was that those fonts were not of the format supported by the NEWS workstations.

VFlib version 1 was able to retrieve bitmaps from outline fonts of different file formats and enabled previewers, printer drivers, and localized TEX/LATEX to employ a mixture of various Japanese fonts. VFlib version 1 and its successor, version 2, have been incorporated into many application programs that require Japanese fonts. Here is a partial list of application programs that use VFlib internally:

- xdvik [50]
  A previewer for DVI (device independent) [31] files localized for Japanese. A DVI file is an output file of TEX and contains typeset results.
- dvipsk [14], dvi2ps [12]
  Both convert a Japanese DVI file into a PostScript file.
- Ghostscript localized for Japanese [33]
  Ghostscript [11] is a free PostScript interpreter. It has been modified to display and print Japanese characters with the help of VFlib.
- MagicPoint [37]
  A presentation software program for the X Window System and Unix.
- X-V [46]
  An extension of the X server that uses VFlib in its font system. It can use many additional font formats that are not supported by the standard X servers.
- xfs-VFlib [45]
  The X server can use the font server program xfs to obtain fonts via the network. xfs-VFlib is a modification of xfs incorporating VFlib.

Problems with fonts and font formats are not unique to Japanese. Even today, application programs or window systems sometimes require a font of a specific format that is not available on the operating system. This is especially true of scripts containing non-Latin characters. Developers have to write routines to access font files for each font file format, which demands detailed knowledge of all the font file formats to be supported.

VFlib version 3 is a multi-script version and aims to ease the situation by handling many kinds of font formats and obtaining bitmaps for various scripts in a unified manner. As VFlib version 3 masks the differences between font formats, it helps programmers to easily develop application programs that display or print non-Latin characters. The authors also have developed several useful applications with VFlib version 3. Some of them will be described in the section APPLICATION PROGRAMS.

In order to handle multi-script text, a font module should fulfill these requirements:

First, the representation form of the character code should be at least 24 bits long. The Chinese, Japanese, and Korean (CJK) languages need so many characters that 16 bits are definitely too few. In fact, some font sets have been created specifically to include as many CJK characters as possible. The GT project [43] produced a TrueType font set named *GT mincho* that contains more than 60 thousand characters. The Mojikyo Institute [22] produced another TrueType font set named *Mojikyo* that contains more than 80 thousand characters. These two projects add new characters to their fonts whenever they find characters that are not included in their character sets. Such efforts are necessary to digitize, for example, classical texts containing characters that are no longer used.

There is one thing to be noted about handling such large fonts. Many font file specifications adopt 8-bit or 16-bit sizes for character codes and the number of characters in a single font file is limited. For character sets containing more characters than these limits, it is necessary to slice the character set and store them in multiple font files. Such slicing is troublesome for writing application programs. A font module designed to handle many characters should offer a virtual single font for such a sliced font file set.

Another requirement posed on a multi-script font module is to have flexibility in writing direction. In multi-script text, characters may be aligned from left to right, from right to left, or from top to bottom. For example, English and French are written from left to right, Hebrew and Arabic are written from right to left, and Mongolian and traditional Japanese are written from top to bottom. A multi-script font module should know and accept all possible directionalities.

Some font modules, e.g. FreeType version 1 [47] and T1lib [36], are independent of operating and/or window systems but support only one font file format. There also exist font modules, e.g. the font service in the X Window System, that support multiple font file formats but are dependent on operating and/or window systems. VFlib has the merits of both, as it supports multiple font file formats with a unified interface and is independent of font functions of the operating and/or window systems. Thus, application programs that use VFlib are not only flexible but also portable. This portability is a major advantage of VFlib.

It is worth emphasizing that VFlib is implemented as a library and thus must be linked to application programs. Since the source code of an existing program must be modified to use VFlib, application programs that come only in the binary format cannot benefit from the flexibility and portability of VFlib.

VFLIB : A FONT MODULE

In this section, we first describe the design and implementation of VFlib version 3. Then we explain two prominent features of VFlib: *implicit font specification* and the *code conversion system*.

The Design

VFlib is designed to be simple, provide exactly those features required to print and display multi-script text, and no more. Its primary objective is to obtain glyph images from *fonts* for printing/displaying. VFlib models a *font* as a set of numbered glyphs.

VFlib assumes that the text has already been typeset with all the necessary kerning, ligatures, and character composition in place. Therefore it does not provide typesetting APIs nor functions for kerning, etc. Even if fonts contain information for kerning and/or ligatures, VFlib does not use them.

The common glyph representation form used in VFlib is the *bitmap*. This is because virtually all printers and window systems can draw glyphs represented in the form of a bitmap; outline format, on the other hand, is often device and system dependent.

When text is processed with VFlib, each character is converted to a bitmap and sent to the output devices. This implies that the font files and rasterizer must reside in the computer and not in the output device. Even when an output device, a laser printer for example, has firmware fonts or can download outline font information, VFlib does not utilize them. This approach guarantees that programs using VFlib can obtain the same result no matter what the output device is; printers, window systems, image files, and others all produce the same output (except for resolution). It is true that this approach increases the amount of data transmitted to printers when compared with transmitting font names and character codes. We feel this is not a serious problem because modern printers have a fast interface, such as 100Base-TX Ethernet.

A font module like VFlib should support different font formats in a unified way. Here, *unified* means that application programs can retrieve glyphs by the same function call regardless of the formats of font files. It also means that application programs can use their own character encoding schemes regardless of the encoding schemes of font files. In other words, a font module should hide the font file formats and convert among multiple encoding schemes. To achieve this, VFlib clearly distinguishes the following two concepts:

- *physical* fonts (or font files): data that VFlib uses as input, and
- *logical* fonts: data that VFlib passes to application programs as output.

This distinction makes the process of font handling clear and easy to track. For example, as we have explained above, CJK fonts with more than 60,000 characters are often sliced into multiple *physical* font files but it will be easier to develop application programs if we can treat them as a single *logical* font file. Other examples that can be defined clearly with this distinction include making a composite glyph from multiple characters and substituting one font for another. (See the section Code Conversion System for detail.)

VFlib uses an external database file called *vflibcap* that defines the mapping between physical fonts (or font files) and logical fonts. The details of vflibcap will be described later in this paper.

While we want to access various font types with one unified function call, different procedures are required to obtain glyph images from different font formats. This problem was solved by designing VFlib as a collection of internal modules called *font drivers* and by assigning each font format to one driver. For each invocation of a glyph request function, VFlib selects one of the font drivers and invokes a glyph request function of the driver. The font driver then reads the font files of the assigned format and returns glyph bitmaps. Note especially that VFlib provides an interface function to install new font drivers. Any application program can add new font drivers into VFlib without modifying the core of VFlib.

Font drivers are classified into two categories:

1. rasterizing drivers for a specific font file format, and
2. character code and glyph mapping drivers.

A rasterizing driver opens zero or more files and generates glyph bitmaps of characters according to the specification of the opened font files.

A mapping driver converts character codes, obtains glyph bitmaps from other font drivers, and then modifies obtained glyph bitmaps. Mapping drivers can be used for various purposes, including:

- To generate a single logical font by recombining a set of sliced font files that collectively represent one character set (useful for CJK characters)
- To generate a logical font for vertical text display by modifying the metrics of a horizontally designed font.
- To make font files in arbitrary format as base fonts used in a TeX Virtual Font.

As the data type for character codes, VFlib uses the `long` type whose size is 32 bits or more (in most systems) to reserve enough space for all possible characters.

In order to support directionality other than left-to-right, the movement of the reference point is described by a vector, i.e., two values that represent the horizontal and vertical movement. As both values can be less than zero, the module is able to represent any movement of the reference point, namely the writing directionality.

The Implementation

Figure 1 shows the internal structure of VFlib and its relations to application programs and to font files. In addition to the VFlib library, we have also developed a stand-alone font server program called vflserver. vflserver can provide glyph bitmaps to application programs, both locally and via a network.

VFlib consists of two parts:

1. The library (`libVFlib.a` or `libVFlib.so`)
   This library provides functions written in C. Application programs link with this library in order to use VFlib functions.

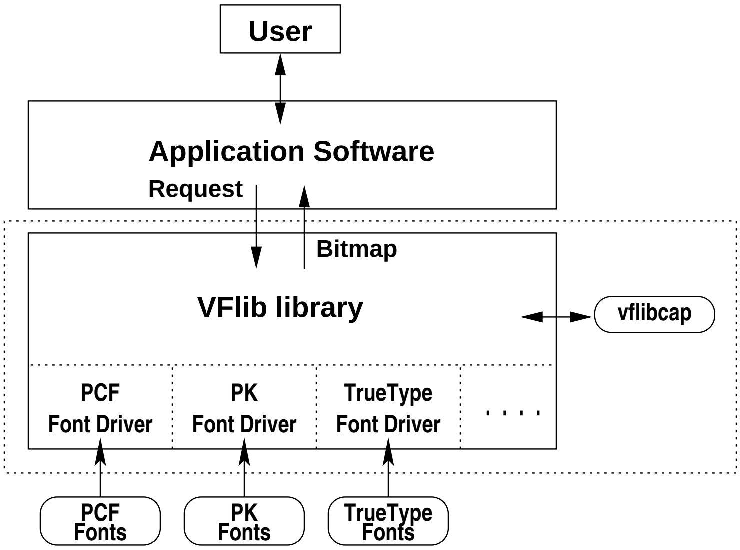

Figure 1. Internal structure of VFlib

2. The font database (`vflibcap`)
   This database defines fonts and their properties such as point size and font file format.

We describe these two parts, one by one, below.

VFlib library

Some basic functions of VFlib are explained below. Here we concentrate on the functions required for glyph image retrieval but VFlib has other functions, including facilities for obtaining font metrics, rasterizing outline data, and composing multiple glyph images.

VFlib has two methods for accessing fonts and specifying font size because output devices are classified into two categories, i.e., low resolution devices such as CRT displays and high resolution devices such as laser printers. For low resolution devices, VFlib uses pixels; for high resolution devices, it uses physical size and device resolution.

- `int VF_Init(char *`*vflibcap*`, char *`*vars*`)`
  This function initializes VFlib. The first argument *vflibcap* is the filename of the vflibcap font database file. A vflibcap file may contain variables whose values are fixed at runtime. The second argument of this function, *vars*, is the list of such variables and their values.
  If VFlib is initialized successfully, a non-negative integer is returned. Otherwise, a negative integer is returned.

- `int VF_OpenFont1(char *font_name, double dpi_x, double dpi_y, double point, double mag_x, double mag_y)`
  This function opens a font named *font_name* for high resolution devices. The next two arguments *dpi_x* and *dpi_y* set horizontal and vertical resolutions of the target device in dpi (dot per inch). *point* defines the font size in points. If this value is positive, the font is scaled to the given size; if negative, the default font size is used. The last two arguments *mag_x* and *mag_y* are the horizontal and vertical magnification factors.
  If the requested font is opened successfully, a non-negative integer (called a *font identifier* and used for future font operations) is returned. Otherwise, a negative integer is returned.
- `int VF_OpenFont2(char *font_name, int pixel, double mag_x, double mag_y)`
  This function opens a font for low resolution devices and returns a font identifier if successful. This function is almost identical to `VF_OpenFont1()` but fonts are scaled by pixel size.
- `VF_BITMAP VF_GetBitmap1(int fid, long code, double mag_x, double mag_y)`
  This function returns a glyph bitmap of a font whose font identifier is *fid* and whose code point is *code*. For the data type `VF_BITMAP`, see below. The parameter *fid* must be a font identifier returned by a previous call to `VF_OpenFont1()`. The size of the glyph to be obtained can be magnified or shrunk by the arguments *mag_x* and *mag_y*.
- `VF_BITMAP VF_GetBitmap2(int fid, long code, double mag_x, double mag_y)`
  This function is similar to `VF_GetBitmap1()` but returns a glyph bitmap of a font for low resolution devices.

The data type `VF_BITMAP` represents a bitmap image, the reference point of the character in the image, and the vector to the next reference point. It is defined as follows:

```
struct vf_s_bitmap {
  int             bbx_width, bbx_height; /* in pixels */
  int             off_x, off_y;          /* in pixels */
  int             mv_x,  mv_y;           /* in pixels */
  unsigned char*  bitmap;
  int             raster;
};
typedef struct vf_s_bitmap*  VF_BITMAP;
```

The members `bbx_width` and `bbx_height` define the width and height of a glyph[†]. The members `off_x` and `off_y` specify the vector from the current reference point to the left top corner of the current glyph. The members `mv_x` and `mv_y` specify the vector from the current reference point to the reference point of the next glyph. The member `bitmap` is the bitmap information itself. The member `raster` defines the number of bytes of `bitmap` per raster.

The following is some example code to obtain and display glyphs. In this example, each character is assumed to be represented by one byte.

[†]For fonts that have multiple metrics (e.g. in horizontal and vertical typesetting), VFlib selects one of them. Unfortunately, the current implementation does not support a metrics selection feature; metrics in horizontal typesetting are selected in font drivers.

```
VF_BITMAP  bm;                          /* pointer to bitmap (glyph) object */
int   fid;                                              /* font identifier */
int   x, y;                                             /* reference point */
char  *p;                                                       /* pointer */
static char str[] = "Hello, World!";                        /* sample text */
extern void display();                       /* character display function */


if (VF_Init("/usr/local/lib/vflibcap", NULL) < 0){  /* Initialize VFlib */
  printf("initialization error.\n");
  exit(0);
}

fid = VF_OpenFont1("timR24", 72, 72, -1, 1, 1);           /* Open a font */

x = y = 0;                                        /* Initial reference point */
for (p = str; *p != '\0'; p++){
  bm = VF_GetBitmap1(fid, (long)*p, 1, 1);                  /* Get a glyph */
  display(bm, x, y);                                /* Display bm at (x, y) */
  x += bm->mv_x;                                    /* Move reference point */
  y += bm->mv_y;
  VF_FreeBitmap(bm);                               /* ... and release a glyph */
}
```

Font database: vflibcap

When opening a font, a logical font name is used. In order to resolve logical font names, VFlib refers to a database file called *vflibcap*. A vflibcap file contains definitions of logical font names. The definition of a logical font specifies how to manipulate the font, e.g. the name of the font driver, the font files to be used, the default font size, etc. The database file is selected at runtime and users can select the one most suitable for each application program and for each execution environment. Figure 2 shows a simple example of a vflibcap file.

The directive `define-default` sets default values of fonts and parameters for a *font class*. A font class is the service unit provided by the font driver that is responsible for a specific font format. When the font class is set to the special value `VFlib`, it sets the global parameters of the font module VFlib. In this case, the property `uncompression-programs` declares the relation between a filename extension and its uncompression program; in the example of Figure 2, it declares that files whose name ends in `.Z` and `.gz` should be uncompressed by the commands `zcat` and `gzip -cd`, respectively. When the font class is other than `VFlib` (`bdf`, `pcf`, etc.) it defines the file compression type and the font directories for that particular font class. (A font directory terminated by two slashes `//` means that font files are searched recursively under the given directory.)

The directive `define-font` specifies a logical font. In Figure 2, two logical fonts, namely `timR24` and `timR18`, are defined. The first property of `timR24` is `font-class`. It specifies that


```
(define-default  VFlib    ;; Parameters for VFlib core
  (uncompression-programs  (".Z" "zcat") (".gz" "gzip -cd")))
(define-default  bdf      ;; Parameters for BDF font class
  (compression-extensions ".gz" ".Z")
  (font-directories  "/usr/local/share/fonts/X11//"
                     "/usr/local/share/emacs/fonts/bdf/"))
(define-default  pcf      ;; Parameters for PCF font class
  (font-directories  "/usr/X11R6/lib/X11/fonts//"
                     "/usr/local/share/emacs/fonts/")
  (compression-extensions ".gz" ".Z"))

(define-font  timR24    ;; Definition of font "timR24" (BDF format)
  (font-class bdf)
  (dpi 300) (point-size 24) (pixel-size 24)
  (font-file "timR24.bdf"))
(define-font  timR18    ;; Definition of font "timR18" (PCF format)
  (font-class pcf)
  (dpi 300) (point-size 18) (pixel-size 18)
  (font-file "timR18.pcf"))
```

Figure 2. A sample vflibcap file

```
(define-default  VFlib    ;; Parameters for VFlib core
  (uncompression-programs  (".Z" "zcat") (".gz" "gzip -cd")))
(define-default  ttf      ;; Parameters for TrueType class
  (font-directories  "/usr/local/share/fonts/TTF//"))

(define-font  timR24    ;; Definition of font "timR24"
  (font-class ttf)
  (dpi 300) (point-size 24) (pixel-size 24)
  (font-file "timR24.ttf"))
(define-font  timR18    ;; Definition of font "timR18"
  (font-class ttf)
  (dpi 300) (point-size 18) (pixel-size 18)
  (font-file "timR18.ttf"))
```

Figure 3. Another sample vflibcap file

`timR24` is a BDF format font. Thus the driver for the `bdf` font class is responsible for the `timR24` logical font. Other properties set the default font size and the font file to be used: the default device resolution is 300 dpi, the default font size is 24 points for high resolution mode and 24 pixels for low resolution mode and the font file name is `timR24.bdf`.

Similarly, the logical font `timR18` is handled by the font driver for the `pcf` font class, the default device resolution is 300 dpi, the default font size is 18 points for high resolution mode and 18 pixels for low resolution mode, and the font file name is `timR18.pcf`.

```
(define-default  VFlib    ;; Parameters for VFlib core
  ; For compressed font files...
  (uncompression-programs  (".Z" "zcat") (".gz" "gzip -cd"))
  ; Invoke the PCF font driver to search implicit fonts.
  (implicit-font-classes  pcf))

(define-default  pcf      ;; Parameters for PCF font class
  (font-directories  "/usr/X11R6/lib/X11/fonts//"
                     "/usr/local/share/emacs/fonts/")
  (compression-extensions ".gz" ".Z"))
```

Figure 4. An example vflibcap file that uses only implicit fonts

Figure 3 is another vflibcap file that employs TrueType font files to define the font classes `timR24` and `timR18`. When this vflibcap file is used as the database, application programs will use TrueType font files.

As VFlib automatically adds a file name extension given in `define-default` and searches for a compressed file, font definitions do not need to specify compressed font files explicitly. When a compressed file is found, VFlib first runs the uncompression program, which is also specified in the font class definition, then reads in the uncompressed font file.

Implicit Font Specification

Since there are many font files in systems such as the X Window System and TEX, it is difficult and troublesome to list them all in a vflibcap file. If such a complete list is required, we must update vflibcap files each time we add new fonts; also, it would be hard to find mistakes in vflibcap such as missing entries and entries for non-existing fonts. VFlib has a mechanism to simplify the contents of a vflibcap file. This mechanism makes it possible to specify the fonts that fulfill certain conditions without giving explicit file names in a vflibcap file. Such fonts are called *implicit fonts*. On the other hand, the font files explicitly listed in a vflibcap file (as in the above examples) are called *explicit fonts*. When an application program requests VFlib to open a font, VFlib first tries to open the requested font as an explicit font; if that fails, VFlib retries to open the requested font as an implicit font. Application programs specify an implicit font by a font file name or by a part of a font file name.

Figure 4 is a vflibcap file with no "`define-font`" directive, which means it defines only implicit fonts and no explicit fonts.

As in the previous examples, the directive "`define-default VFlib`" defines default values for all font classes. The property `implicit-font-classes` specifies the font classes whose members are regarded as implicit fonts. In this example, the PCF fonts are declared to be implicit fonts. The second directive "`define-default pcf`" sets parameters for the PCF font class. The first property `font-directories` specifies the directories in which files are searched.

```
(define-default VFlib
  ; Invoke PK font driver to search implicit fonts.
  (implicit-font-classes  pk)
  ; Initialization parameters for 'kpathsea' library.
  (use-kpathsea           "yes")
  (kpathsea-mode          "cx")
  (kpathsea-dpi           300)
  (kpathsea-program-name  "/usr/local/bin/xgdvi"))

(define-default pk
  ; Font file directory. "TEXMF" is a special name to search
  ; font files in TeX directory by 'kpathsea' library.
  (font-directories  "TEXMF" ".")
  ; Generate PK font file from metafont source on-the-fly,
  ; if PK font file does not exist.
  (make-missing-glyph "yes"))
```

Figure 5. A complex vflibcap file using implicit fonts

A Complex Example with Implicit Fonts

Figure 5 is a simple but powerful vflibcap file. It defines TEX PK fonts as implicit fonts and invokes other programs on-the-fly if necessary.

PK fonts are stored in special directory trees and it is impractical to list all PK font directories in a vflibcap file. Fortunately, there is a library called kpathsea that looks for requested PK font in the directory trees and returns the full pathname of the font file if it is found.

The first directive in Figure 5 makes VFlib invoke kpathsea to search for implicit PK fonts. This directive also defines necessary parameters to initialize kpathsea.

The property "`font-directories`" in the second directive specifies the PK font directories in a special way. Here, the string "`TEXMF`" is a special keyword to invoke kpathsea to search a font file in the PK directory trees. The second string "`.`" represents the current directory.

The property "`make-missing-glyph yes`" in the second directive means that VFlib invokes the TEX utilities METAFONT and GFtoPK to create necessary PK fonts on-the-fly if the requested font is not found in the PK directory trees. Although frequently used PK fonts are installed with TEX, certain documents require PK fonts that are not preinstalled. Usually the user must manually generate necessary PK fonts in that case but with the property setting "`make-missing-glyph yes`" VFlib does the job instead.

VFlib with the vflibcap file shown in Figure 5 behaves as follows when it is requested to open a font:

1. Since there is no explicit font, VFlib tries to open the requested font as an implicit font.
2. Since the PK font class is the only font class for implicit fonts, VFlib calls an implicit font search method of the PK font driver.

Prepared using speauth.cls

3. The PK font driver searches the font in each directory specified by the property "`font-directories`":

   (a) First, the PK font driver invokes kpathsea to find the font file. If the file is found, VFlib opens it.
   (b) If it is not found there, the driver checks the current directory and opens the file if found.
   (c) If the font is still not found, the PK font driver invokes METAFONT and then GFtoPK to generate a new PK font file. If a font file is successfully generated, it is opened.

4. If none of the above succeeds, VFlib fails to open the requested font.

Code Conversion System

Many characters belong to more than one character set. Suppose that the font of a specific character is not available in the requested encoding, e.g. ISO 8859-2 but is available in another, e.g. Unicode. In this case the font module would utilize the font in the Unicode encoding as if it were in the ISO 8859-2 encoding, provided it is equipped with a code conversion facility.

Such a facility is useful especially for accessing TrueType fonts because TrueType fonts make the encoding scheme invisible to users. For example, TrueType font files for Japanese Kanji characters (JIS X 0208) often adopt Unicode or Shift JIS (MS Kanji) encoding but users cannot see which encoding is really used. In order to retrieve bitmaps from a TrueType font file, an application program has to find out the encoding scheme of the font file and convert the character codes to its own encoding if necessary. It is, however, neither convenient nor efficient to write such a routine only for TrueType fonts.

VFlib provides a general character code conversion system called *CCV*. If an application program uses VFlib and uses CCV to convert, for example, the JIS encoding to the encoding scheme of the available TrueType font, it can use TrueType Japanese fonts just like other Japanese JIS encoded fonts.

Code conversion is specified by a pair consisting of (1) a character set and its internal encoding scheme and (2) the converted character set and the encoding schemes that are externally visible (i.e., character code visible to application programs).

Conversion rules are implemented either by VFlib's internal functions or by external conversion tables. While internal functions are hard-coded and difficult to modify, external conversion tables are specified in font definitions by the directive `define-default` and are easy to customize. Internal functions are currently restricted to conversions that require simple arithmetic calculations.

The current version of CCV has code conversion rules shown in Table II. Moreover, users can define their own conversion rules.

Table II. Code conversion rules supported by the current CCV (In this table "Unicode" means the BMP encoded in UCS-2)

External (application program) side		Internal (font file) side	
Character set	Encoding scheme	Character set	Encoding scheme
ISO 8859-{1 – 10, 13 – 15}		Unicode	
ISO 8859-5		KOI8-R	
KOI8-R		{Unicode, ISO 8859-5}	
JIS X 0201		Unicode	
JIS X 0208	{ISO 2022, Shift JIS}	Unicode	
JIS X 0208	{ISO 2022, EUC}	JIS X 0208	Shift JIS
JIS X 0208	{Shift JIS, EUC}	JIS X 0208	ISO 2022
JIS X 0212	ISO 2022	Unicode	
KSC 5601	ISO 2022	Unicode	
KSC 5601	{ISO 2022, EUC}	KSC 5601	{ISO 2022, EUC}
KSX 1001	ISO 2022	Unicode	
GB 2312	ISO 2022	Unicode	
GB 2312	{ISO 2022, EUC}	GB 2312	{ISO 2022, EUC}
GB 12345	ISO 2022	Unicode	
Big 5		Unicode	
CNS 11643 Plane 1, 2	ISO 2022	Unicode	

APPLICATION PROGRAMS

We have developed several VFlib dependent application programs. In this section we describe three of them: ps-multi, xgdvi, and ctext2pgm. From these implementation experiments, we confirmed that VFlib makes it easy and straightforward to implement application programs as it hides the details of font files such as the format and the encoding.

ps-multi

The text editor Emacs is widely used throughout the world and its version 20 or later can handle multi-script text. The Emacs library ps-print generates PostScript files containing multi-script text but it can utilize only built-in printer fonts and external font files in BDF format.

We have enhanced ps-print and named it ps-multi. Since ps-multi uses VFlib as the font module, it can use fonts in various formats to generate multi-script PostScript files. Of course, the PostScript generated by ps-multi can be printed with a standard PostScript printer.

Figure 6 shows an example of multilingual and multi-script text printing. In this example, the Latin script is printed in Courier, which is built into the PostScript printer, the Thai and Japanese scripts are printed using TrueType fonts, and other scripts are printed using BDF fonts. All scripts but the Latin are printed using VFlib. As shown in this figure, if fonts of better quality are available for some scripts, they can be used while resorting to BDF fonts for other scripts.

sample.txt **1/1**
/project/vflib/paper/spe–vflib–psprint/ Sep 17 1999

```
-------------------------------------
   'Hello' in various languages
-------------------------------------
English(Hello), German(Grüß Gott),

Greek(Γειά σας), Russian(Здравствуйте),

Thai(สวัสดีครัป), Hindi(नमस्ते, नमस्कार ।),

Tibetan(བཀྲ་ཤིས་བདེ་ལེགས།།),

Chinese(你好), Japanese(今日は),

Korean (안녕하세세요)
```

Figure 6. Example of multi-lingual text printed with ps-multi

ps-multi consists of the following three parts:

1. An interface to VFlib:
   Emacs has a general interface for communicating with external programs. With this interface Emacs sends text to external programs and receives the output from them. ps-multi customizes the interface so as to communicate with vflserver (the stand-alone server version of VFlib) so that Emacs can get bitmap data of a specific character in a specific font.
2. A font name database:
   The variable `ps-mule-font-info-database` works as the database for ps-multi. It associates each character set in a buffer with the corresponding font. Each font is identified by its type and its name. If the font type is `vflib`, the font name is interpreted as an entry in a vflibcap file. For the VFlib users' sake, the predefined constant `ps-mule-font-info-database-vflib` is provided.
3. A new font type driver:

Prepared using speauth.cls

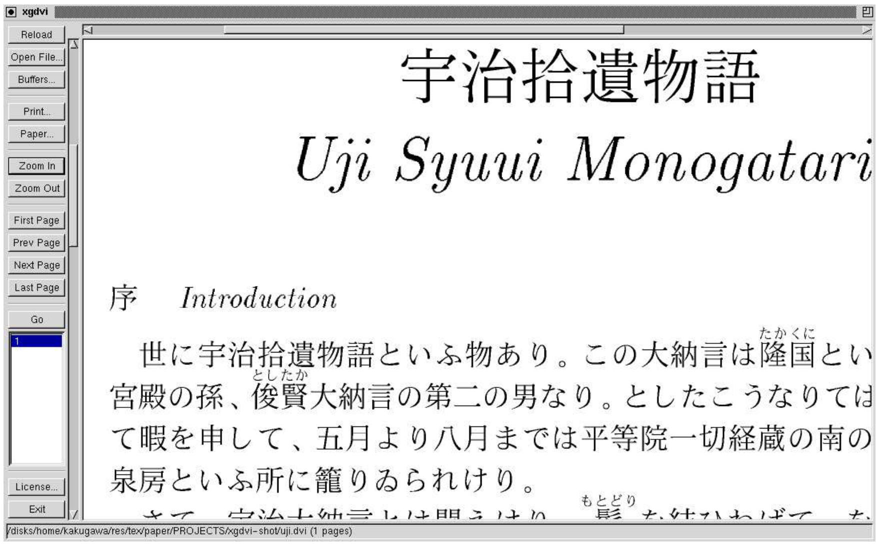

Figure 7. Preview window using PK (English) and TrueType (Japanese) fonts

Each font type is associated with a function to obtain the glyph images of that font. When the font type is `vflib`, Emacs sends requests to vflserver to open the specified font and to return the bitmap data of the requested characters.

xgdvi

We have developed a new TEX DVI file previewer xgdvi. It is written using the DVI interpreter library DVIlib [27], the font module VFlib, and the GUI toolkit gtk+ [17]. Most of the newly written code is for the user interface. Figure 7 and Figure 8 show two display snapshots generated by xgdvi.

In these two figures, the same DVI file is displayed with different sets of fonts. Figure 7 is a snapshot where English is displayed with PK fonts while the Japanese is displayed with a TrueType font encoded in Shift JIS. In Figure 8, English is displayed with a Type 1 font while the Japanese is displayed with a PCF font encoded in a variant of ISO 2022.

These two snapshots were prepared by specifying two different vflibcap files as the font database. There was no need to change text files or font names. Actually, xgdvi does not know which physical font files are being used.

As described above, VFlib hides the difference of font encodings from application programs. Therefore, xgdvi can use TrueType fonts encoded using Shift JIS even though it assumes an ISO 2022 style encoding for the Japanese script.

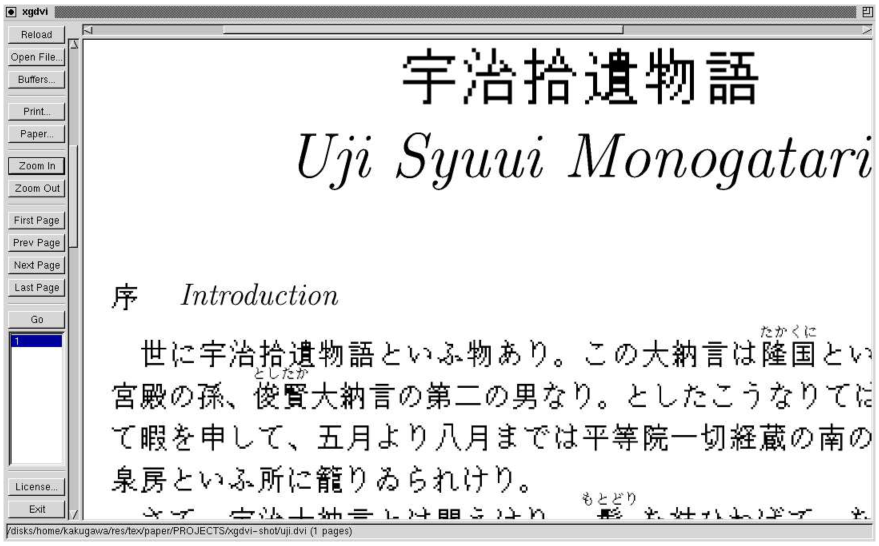

Figure 8. Preview window using Type 1 (English) and PCF (Japanese) fonts

Figure 9. A sample Hebrew document by ε-TEX

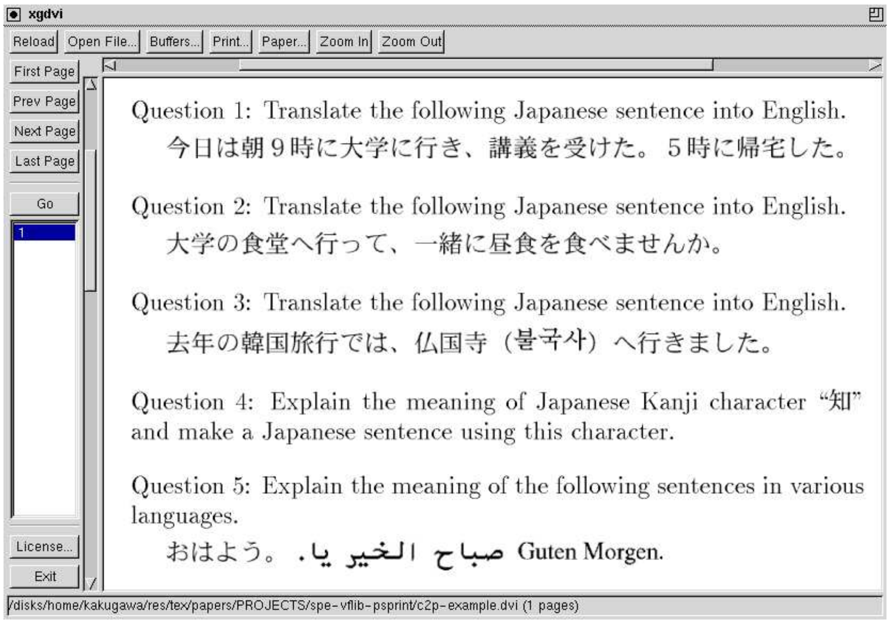

Figure 10. Example of TEX document with ctext2pgm outputs

While most previewers and printer drivers for Japanese TEX or multilingual TEX implement Latin fonts and CJK fonts separately, xgdvi and other DVI interpreters with VFlib can use GF, PK, Type 1, and TrueType fonts, regardless of character sets. This means that font handling routines will be simple and fast.

xgdvi itself does not handle bi-directionality because it is not necessary as long as it is used with ε-TEX [15]. ε-TEX is an extension of TEX for multilingual text. It typesets bi-directional documents by reversing the text order of the right-to-left part in the right-to-left typesetting mode [16]. As a result, its output can always be rendered from left to right and neither previewers nor printer drivers need to be aware of bi-directionality. Figure 9 is an example of right-to-left text typeset by ε-TEX and displayed by xgdvi. (The document shown in the Figure is taken from the Hebrew package for ε-TEX. [4])

ctext2pgm

Another application program is ctext2pgm. It creates a bitmap image file from a multilingual plain text file encoded in the compound text (ctext) scheme. It can handle an input file that contains scripts of different writing directions intermixed. That means, Arabic and Hebrew, which are written from right to left, can appear in a file with European and/or CJK scripts. Such files are difficult to display in a standard environment, but ctext2pgm can convert them into image files that can appear on an ordinary bitmap screen. Currently, it supports PBM (Portable Bitmap), PGM (Portable Graymap), and EPS (Encapslated PostScript) formats for output image file.

Figure 10 is an output of TEX. Sequences of Japanese, Korean and Arabic text are first converted to EPS files by ctext2pgm and then imported into the TEX document as figures. The last line in the figure is a single EPS file containing both left-to-right and right-to-left scripts. Font files used for Japanese and Korean are TrueType fonts while a font file for Arabic is a 24 pixel bitmap font in PCF.

CONCLUSION

We have described the font module VFlib and three application programs for multilingual documents. VFlib enables application programs to make use of fonts of different formats and different encoding schemes in a unified manner.

From our experiences implementing multilingual or multi-script application programs, we believe that it is very important to be able to print and/or display text with already available fonts. It is much better to have a low quality output with bitmap fonts than to have no output at all. As VFlib hides all the details of font files from application programs, it is easy to cope with changes in the available fonts. When a new outline font becomes available, users can have higher quality output with minimal changes.

In this world, there are still scripts and character sets that are not widely used and not well supported by computers. VFlib fills one of the gaps that bar users of such scripts from the benefit of information technology.

We admit that Unicode may be enough for average users; adopting Unicode compliant systems with high quality TrueType, Type 1 or OpenType fonts would be the best solution for them. However, to display and print characters not included in Unicode or other standards, a mechanism to extend supported character sets and fonts like VFlib is necessary in operating/window/printing systems.

Future Work

The current version of VFlib lacks one important feature; a font selection mechanism by properties, e.g. character set name, font family, font size, etc. We are planning to implement such a feature in later versions.

Distribution

VFlib is written in C and its size is about 34 thousand lines. It is distributed as free software under the terms of GNU Library General Public License (LGPL). VFlib has been tested on FreeBSD, Linux, HP-UX, Solaris, and SunOS, but it should not be difficult to port to other Unix-like systems. For the latest information and the source code, access `http://TypeHack.aial.hiroshima-u.ac.jp/VFlib/`.

ACKNOWLEDGMENT

The authors would like to thank Martin Buchholz and Sandy Wambold for proofreading this article.

REFERENCES

1. Adobe Systems Inc. *Adobe Type 1 Font Format, version 1.1.* Addison-Wesley, 1990.
2. Adobe Systems Inc. Glyph Bitmap Distribution Format (BDF) specification version 2.2. http://www.adobe.com/supportservice/devrelations/PDFS/TN/5005.BDF_Spec.pdf, 1993.
3. Karl Berry. Kpathsea: a library for path searching. http://www.tug.org/kpathsea/.
4. Johannes L. Braams. Hebrew package for babel system and LaTeX2ε. ftp://ftp.cc.huji.ac.il/tex/macros/latex_macros/babel/.
5. Apple Computer. TrueType reference manual. http://fonts.apple.com/TTRefMan/index.html.
6. The Unicode consortium. The Unicode standard. http://www.unicode.org/.
7. Adobe Coop. OpenType specification. http://partners.adobe.com/asn/developer/opentype/main.html.
8. ASCII Coop. pTeX. http://www.ascii.co.jp/pb/ptex/.
9. Microsoft Coop. OpenType specification. http://www.microsoft.com/typography/tt/tt.htm.
10. Microsoft Coop. TrueType specification. http://www.microsoft.com/typography/tt/tt.htm.
11. L. Peter Deutsch. GNU Ghostscript. http://www.gnu.org/directory/ghostscript.html.
12. dvi2ps. ftp://ftp.math.s.chiba-u.ac.jp/tex/dvi2ps-2.0j.tar.gz. in FreeBSD ports: japanese/dvi2ps.
13. dvipsk. http://www.ctan.org/tex-archive/dviware/dvipsk/.
14. Japanese localized dvipsk. http://math.shinshu-u.ac.jp/˜nakayama/howto/tex/dvipsk-vflib.html. in FreeBSD ports: japanese/dvipsk-vflib.
15. ε-TeX. http://www.ctan.org/tex-archive/systems/e-tex/.
16. The ε-TeXReference Manual. http://www.ctan.org/tex-archive/systems/e-tex/v2/doc/etex_man.ps.
17. Gtk: The GIMP toolkit. http://www.gtk.org/.
18. Ken'ichi Handa, Mikiko Nishikimi, and Satoru Tomura. Mule: A MULtilingual Enhancement to GNU Emacs. In *Proceedings of INET '93: International networking conference*, pages GAB 1–9. Internet Society, 1993.
19. Yannis Haralambous and John Plaice. Methods for processing languages with Ω. In *International Symposium on Multilingual Information Processing*, pages 115–128. Electrotechnical Laboratory, March 1997.
20. Yannis Haralambous and John Plaice. Multilingual typesetting with Ω: Case study: Arabic. In *International Symposium on Multilingual Information Processing*, pages 137–154. Electrotechnical Laboratory, March 1997.
21. Roger D. Hersch, editor. *Visual and Technical Aspect of Type.* Cambridge University Press, 1993.
22. The Mojikyo Institute. Mojikyo. http://www.mojikyo.org/.
23. JIS X 0208-1997 Code of the Japanese graphic character set for information interchange. Japanese Standards Association, 1997.
24. JIS X 0212-1990 Code of the supplementary Japanese graphic character set for information interchange. Japanese Standards Association, 1990.
25. JIS X 0213-2000 7-bit and 8-bit double byte coded extended Kanji sets for information interchange. Japanese Standards Association, 2000.
26. Hirotsugu Kakugawa. VFlib — a general font library that supports multiple font formats. In *Proceedings of the 10th European TeX conference (EuroTeX 98)*, pages 221–222, March 1998.
27. Hirotsugu Kakugawa. A device independent DVI interpreter library for various output devices. In *Proceedings of the TUG 2000 Conference*, pages 102–107, Wadham College, Oxford, UK, August 2000.
28. Donald E. Knuth. *The METAFONT book.* Addison-Wesley, 1986.
29. Donald E. Knuth. *METAFONT: The Program.* Addison-Wesley, 1986.
30. Donald E. Knuth. *The TeX book.* Addison-Wesley, 1986.
31. Donald E. Knuth. *TeX: The Program.* Addison-Wesley, 1986.
32. Donald E. Knuth. VPtoVF version 1.5. http://www.ctan.org/tex-archive/systems/knuth/etc/vptovf.web, 1998.

33. Ghostscript with VFlib patch. for example, ftp://ftp.gnu.org/gnu/ghostscript/gs5.50-vflib-1.0.tar.gz. in FreeBSD ports: japanese/vfghostscript55.
34. Ken Lunde. *Understanding Japanese Information Processing.* O'Reilly, 1993.
35. Ken Lunde. *CJKV Information Processing.* O'Reilly, 1999.
36. Rainer Menzner. A library for generating character bitmaps from Adobe Type 1 fonts. http://www.neuroinformatik.ruhr-uni-bochum.de/ini/PEOPLE/rmz/t1lib/t1lib.html.
37. Yoshifumi Nishida. MagicPoint - an X11 based presentation tool. http://www.Mew.org/mgp/, 2000.
38. Mikiko Nishikimi, Ken'ichi Handa, Satoru Tomura, and Naoto Takahashi. Merging Mule with GNU Emacs. In *International Symposium on Multilingual Information Processing*, pages 239–249. Electrotechnical Laboratory, March 1997.
39. Mikiko Nishikimi, Naoto Takahashi, Satoru Tomura, Ken'ichi Handa, Seiji Kuwari, Shin'ichi Mukaigawa, and Tomoko Yoshida. *Creating a multilingual environment.* Prentice Hall, 1996. (in Japanese).
40. Keith Packard. bdftopcf – a converter program from BDF to PCF. in X Window System Version 11 Release 6.
41. John Plaice and Yannis Haralambous. Draft documentation for the Ω system. http://www.ctan.org/tex-archive/systems/omega/omega-doc-1.8.tar.gz, 1998.
42. Man-Chi Pong. Hanzi Bitmap Font (HBF) file format version 1.1. ftp://ftp.ifcss.org/pub/software/info/HBF-1.1.tar.gz, September 1994.
43. The GT Project. http://www.l.u-tokyo-.ac.jp/GT/.
44. Tomas Rokicki. GFtoPK version 2.3. http://www.ctan.org/tex-archive/systems/knuth/mfware/gftopk.web, 1990.
45. Takuya Shiozaki. xfs-VFlib. http://www.imou.to/~AoiMoe/UNIX-at-Random/obsoleted/FreeBSD/xfs-VFlib/.
46. Junji Takagi. VFlib support patch for XFree86 3.3.1. ftp://ftp.riken.go.jp/pub/misc/Japan/X-VFlib/, 1997.
47. FreeType Development Team. The freetype package. http://www.freetype.org/, 1999.
48. The WadaLab vector font in Syotai-Kurabu format. ftp://ftp.eos.hokudai.ac.jp/pub/TeX/fonts/wadalab-vector-font/.
49. The Watanabe vector font in Syotai-Kurabu format. ftp://ftp.eos.hokudai.ac.jp/pub/TeX/fonts/watanabe-vector-font/.
50. xdvik. http://www.ctan.org/tex-archive/dviware/xdvik/.
51. Japanese localized xdvik. http://www.yama-ga.com/xdvi-j/. in FreeBSD ports: japanese/xdvik-vflib.