TEXware

Virtual Fonts: More Fun for Grand Wizards†

Donald Knuth

Many writers to TEXhax during the past year or so have been struggling with interfaces between differing font conventions. For example, there's been a brisk correspondence about mixing oldstyle digits with a caps-and-small-caps alphabet. Other people despair of working with fonts supplied by manufacturers like Autologic, Compugraphic, Monotype, etc.; still others are afraid to leave the limited accent capabilities of Computer Modern for fonts containing letters that are individually accented as they should be, because such fonts are not readily available in a form that existing TEX software understands.

There is a much better way to solve such problems than the remedies that have been proposed in TEXhax. This better way was first realized by David Fuchs in 1983, when he installed it in our DVI-to-APS software at Stanford (which he also developed for commercial distribution by ArborText). We used it, for example, to typeset my article on Literate Programming for *The Computer Journal*, using native Autologic fonts to match the typography of that journal.

I was expecting David's strategy to become widely known and adopted. But alas — and this has really been the only significant disappointment I've had with respect to the way TEX has been propagating around the world — nobody else's DVI-to-X drivers have incorporated anything resembling David's ideas, and TEXhax contributors have spilled gallons of electronic ink searching for answers in the wrong direction.

The right direction is obvious once you've seen it (although it wasn't obvious in 1983): All we need is a good way to specify a mapping from TEX's notion of a font character to a device's capabilities for printing. Such a mapping was called a "virtual font" by the AMS speakers at the TUG meetings this past August. At that meeting I spoke briefly about the issue and voiced my hope that all dvi drivers be upgraded within a year to add a virtual font capability. Dave Rodgers of ArborText announced that his company would make their WEB routines for virtual font design freely available, and I promised to edit them into a form that would match the other programs in the standard TEXware distribution.

The preparation of TEX Version 3 and MF Version 2 has taken me much longer than expected, but at last I've been able to look closely at the concept of virtual fonts. (The need for such fonts is indeed much greater now than it was before, because TEX's new multilingual capabilities are significantly more powerful only when suitable fonts are available. Virtual fonts can easily be created to meet these needs.)

After looking closely at David Fuchs's original design, I decided to design a completely new file format that would carry his ideas further, making the virtual font mechanism completely device-independent; David's original code was very APS-specific. Furthermore I decided to extend his notions so that arbitrary dvi commands (including

---

†: Prof. Knuth, tym razem jeszcze nie pisze specjalnie do biuletynu GUST, jest to przedruk artykułu, który ukazał się w *TUGboat* nr 11, kwiecień 1990 (str. 13–23).

Ponieważ tekst *„Virtual Fonts: More Fun for Grand Wizards"* nie ukazał się w żadnej innej formie pisanej, a dokładniej, jest on oparty na dokumentacji programów TVtoVP i VPtoVF czyli na odpowiednich plikach *.web co oznacza, że za wyjątkiem dwóch stron wstępu resztę można sobie wydrukować samemu — jeżeli się wie co to jest i do czego służy WEB. Ponieważ jednak tylko niewielki odsetek użytkowników systemu TEX korzysta jednocześnie z systemu WEB (pełna nazwa: *The* WEB *System of Structured Documentation*) uznaliśmy za celowe przedrukowanie oryginalnego tekstu prof. Knutha w biuletynie GUST.

Barbara Beeton, szefowa *TUGboat*-a, którą mieliśmy przyjemność poznać na konferencji „Word-wide window on TEX", Aston '93 bez problemów udostępniła nam plik źródłowy artykułu, za co dziękujemy.

Zamiast tłumaczenia w kolejnym tekście przedstawiono kilka przykładów praktycznego wykorzystania fontów wirtualnych.

Red.

rules and even specials) could be part of a virtual font. The new file format I've just designed is called vf; it's easy for dvi drivers to read vf files, because vf format is similar to the pk and dvi formats they already deal with.

The result is two new system routines called VFtoVP and VPtoVF. These routines are extensions of the old ones called TFtoPL and PLtoTF; there's a property-list language called VPL that extends the ordinary PL format so that virtual fonts can be created easily.

In addition to implementing these routines, I've also tested the ideas by verifying that virtual fonts could be incorporated into Tom Rokicki's dvips system without difficulty. I wrote a C program (available from Tom) that converts Adobe afm files into virtual fonts for TEX; these virtual fonts include almost all the characteristics of Computer Modern text fonts (lacking only the uppercase Greek and the dotless j) and they include all the additional Adobe characters as well. These virtual fonts even include all the "composite characters" listed in the afm file, from 'Aacute' to 'zcaron'; such characters are available as ligatures. For example, to get 'Aacute' you type first 'acute' (which is character 19 = ^S in Computer Modern font layout; it could also be character 194 = Meta-B if you're using an 8-bit keyboard with the new TEX) followed by 'A'. Using such fonts, it's now easier for me to typeset European language texts in Times-Roman and Helvetica and Palatino than in Computer Modern! [But with less than an hour's work I could make a virtual font for Computer Modern that would do the same things; I just haven't gotten around to it yet.]

[A nice ligature scheme for dozens of European languages was just published by Haralambous in the November *TUGboat*. He uses only ASCII characters, getting Aacute with the combination <A. I could readily add his scheme to mine, by adding a few lines to my vpl files. Indeed, multiple conventions can be supported simultaneously (although I don't recommend that really).]

Virtual fonts make it easy to go from dvi files to the font layouts of any manufacturer or font supplier. They also (I'm sorry to say) make "track kerning" easy, for people who have to resort to that oft-abused feature of lead-free type.

Furthermore, virtual fonts solve the problem of proofreading with screen fonts or with lowres laserprinter fonts, because you can have several virtual fonts sharing a common tfm file. Suppose, for example, that you want to typeset camera copy on an APS machine using Univers as the ultimate font, but you want to do proofreading with a screen previewer and with a laserprinter. Suppose further that you don't have Univers for your laserprinter; the closest you have is Helvetica. And suppose that you haven't even got Helvetica for your screen, but you do have cmss10. Here's what you can do: First make a virtual property list (vpl) file univers-aps.vpl that describes the high-quality font of your ultimate output. Then edit that file into univers-laser.vpl, which has identical font metric info but maps the characters into Helvetica; similarly, make univers-screen.vpl, which maps them into cmss10. Now run VPtoVF on each of the three vpl files. This will produce three identical tfm files univers.tfm, one of which you should put on the directory read by TEX. You'll also get three distinct vf files called univers.vf, which you should put on three different directories—one directory for your DVI-to-APS software, another for your DVI-to-laserwriter software, and the third for the DVI-to-screen previewer. Voilà.

So virtual fonts are evidently quite virtuous. But what exactly are virtual fonts, detail-wise? Appended to this message are excerpts from VFtoVP.web and VPtoVF.web, which give a complete definition of the vf and vpl file formats.

I fully expect that all people who have implemented dvi drivers will immediately see the great potential of virtual fonts, and that they will be unable to resist installing a vf capability into their own software during the first few months of 1990. (The idea is this: For each font specified in a dvi file, the software looks first in a special table to see if the font is device-resident (in which case the tfm file is loaded, to get the character widths); failing that, it looks for a suitable gf or pk file; failing that, it looks for a vf file, which may in turn lead to other actual or virtual files. The latter files should not be loaded immediately, but only on demand, because the process is recursive. Incidentally, if no resident or gf or pk or vf file is found, a tfm file should be loaded as a last resort, so that the characters can be left blank with appropriate widths.)

An Excerpt from `VFtoVP.web`

6. Virtual fonts. The idea behind VF files is that a general interface mechanism is needed to switch between the myriad font layouts provided by different suppliers of typesetting equipment. Without such a mechanism, people must go to great lengths writing inscrutable macros whenever they want to use typesetting conventions based on one font layout in connection with actual fonts that have another layout. This puts an extra burden on the typesetting system, interfering with the other things it needs to do (like kerning, hyphenation, and ligature formation).

These difficulties go away when we have a "virtual font," i.e., a font that exists in a logical sense but not a physical sense. A typesetting system like TEX can do its job without knowing where the actual characters come from; a device driver can then do its job by letting a VF file tell what actual characters correspond to the characters TEX imagined were present. The actual characters can be shifted and/or magnified and/or combined with other characters from many different fonts. A virtual font can even make use of characters from virtual fonts, including itself.

Virtual fonts also allow convenient character substitutions for proofreading purposes, when fonts designed for one output device are unavailable on another.

7. A VF file is organized as a stream of 8-bit bytes, using conventions borrowed from DVI and PK files. Thus, a device driver that knows about DVI and PK format will already contain most of the mechanisms necessary to process VF files. We shall assume that DVI format is understood; the conventions in the DVI documentation (see, for example, *TEX: The Program*, part 31) are adopted here to define VF format.

A preamble appears at the beginning, followed by a sequence of character definitions, followed by a postamble. More precisely, the first byte of every VF file must be the first byte of the following "preamble command":

pre

247 $i[1]$ $k[1]$ $x[k]$ $cs[4]$ $ds[4]$. Here i is the identification byte of VF, currently 202. The string x is merely a comment, usually indicating the source of the VF file. Parameters cs and ds are respectively the check sum and the design size of the virtual font; they should match the first two words in the header of the TFM file, as described below.

After the *pre* command, the preamble continues with font definitions; every font needed to specify "actual" characters in later *set_char* commands is defined here. The font definitions are exactly the same in VF files as they are in DVI files, except that the scaled size s is relative and the design size d is absolute:

fnt_def1

243 $k[1]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $0 \le k < 256$.

fnt_def2

244 $k[2]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $0 \le k < 65536$.

fnt_def3

245 $k[3]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $0 \le k < 2^{24}$.

fnt_def4

246 $k[4]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $-2^{31} \le k < 2^{31}$.

These font numbers k are "local"; they have no relation to font numbers defined in the DVI file that uses this virtual font. The dimension s, which represents the scaled size of the local font being defined, is a *fix_word* relative to the design size of the virtual font. Thus if the local font is to be used at the same size as the design size of the virtual font itself, s will be the integer value 2^{20}. The value of s must be positive and less than 2^{24} (thus less than 16 when considered as a *fix_word*). The dimension d is a *fix_word* in units of printer's points; hence it is identical to the design size found in the corresponding TFM file.

define *id_byte* = 202

⟨ Globals in the outer block 7 ⟩ ≡
vf_file: **packed file of** 0 . . 255;

See also sections 10, 12, 20, 23, 26, 29, 30, 37, 42, 49, 51, 54, 67, 69, 85, 87, 111, and 123.
This code is used in section 2.

8. The preamble is followed by zero or more character packets, where each character packet begins with a byte that is < 243. Character packets have two formats, one long and one short:

long_char
242 *pl*[4] *cc*[4] *tfm*[4] *dvi*[*pl*]. This long form specifies a virtual character in the general case.
short_char0 . . *short_char241* *pl*[1] *cc*[1] *tfm*[3] *dvi*[*pl*]. This short form specifies a virtual character in the common case when $0 \le pl < 242$ and $0 \le cc < 256$ and $0 \le tfm < 2^{24}$.

Here *pl* denotes the packet length following the *tfm* value; *cc* is the character code; and *tfm* is the character width copied from the TFM file for this virtual font. There should be at most one character packet having any given *cc* code.

The *dvi* bytes are a sequence of complete DVI commands, properly nested with respect to *push* and *pop*. All DVI operations are permitted except *bop*, *eop*, and commands with opcodes ≥ 243. Font selection commands (*fnt_num0* through *fnt4*) must refer to fonts defined in the preamble.

Dimensions that appear in the DVI instructions are analogous to *fix_word* quantities; i.e., they are integer multiples of 2^{-20} times the design size of the virtual font. For example, if the virtual font has design size 10 pt, the DVI command to move down 5 pt would be a *down* instruction with parameter 2^{19}. The virtual font itself might be used at a different size, say 12 pt; then that *down* instruction would move down 6 pt instead. Each dimension must be less than 2^{24} in absolute value.

Device drivers processing VF files treat the sequences of *dvi* bytes as subroutines or macros, implicitly enclosing them with *push* and *pop*. Each subroutine begins with $w = x = y = z = 0$, and with current font f the number of the first-defined in the preamble (undefined if there's no such font). After the *dvi* commands have been performed, the h and v position registers of DVI format and the current font f are restored to their former values; then, if the subroutine has been invoked by a *set_char* or *set* command, h is increased by the TFM width (properly scaled)—just as if a simple character had been typeset.

define *long_char* = 242 { VF command for general character packet }
define *set_char_0* = 0 { DVI command to typeset character 0 and move right }
define *set1* = 128 { typeset a character and move right }
define *set_rule* = 132 { typeset a rule and move right }
define *put1* = 133 { typeset a character }
define *put_rule* = 137 { typeset a rule }
define *nop* = 138 { no operation }
define *push* = 141 { save the current positions }
define *pop* = 142 { restore previous positions }
define *right1* = 143 { move right }
define *w0* = 147 { move right by w }
define *w1* = 148 { move right and set w }
define *x0* = 152 { move right by x }
define *x1* = 153 { move right and set x }
define *down1* = 157 { move down }

define *y0* = 161 { move down by y }
define *y1* = 162 { move down and set y }
define *z0* = 166 { move down by z }
define *z1* = 167 { move down and set z }
define *fnt_num_0* = 171 { set current font to 0 }
define *fnt1* = 235 { set current font }
define *xxx1* = 239 { extension to DVI primitives }
define *xxx4* = 242 { potentially long extension to DVI primitives }
define *fnt_def1* = 243 { define the meaning of a font number }
define *pre* = 247 { preamble }
define *post* = 248 { postamble beginning }
define *improper_DVI_for_VF* ≡ 139, 140, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255

9. The character packets are followed by a trivial postamble, consisting of one or more bytes all equal to *post* (248). The total number of bytes in the file should be a multiple of 4.

And Here's an Extract from `VPtoVF.web`

5. Property list description of font metric data. The idea behind VPL files is that precise details about fonts, i.e., the facts that are needed by typesetting routines like TEX, sometimes have to be supplied by hand. The nested property-list format provides a reasonably convenient way to do this.

A good deal of computation is necessary to parse and process a VPL file, so it would be inappropriate for TEX itself to do this every time it loads a font. TEX deals only with the compact descriptions of font metric data that appear in TFM files. Such data is so compact, however, it is almost impossible for anybody but a computer to read it.

Device drivers also need a compact way to describe mappings from TEX's idea of a font to the actual characters a device can produce. They can do this conveniently when given a packed sequence of bytes called a VF file.

The purpose of VPtoVF is to convert from a human-oriented file of text to computer-oriented files of binary numbers. There's a companion program, VFtoVP, which goes the other way.

⟨ Globals in the outer block 5 ⟩ ≡
vpl_file: *text*;

See also sections 21, 24, 27, 29, 31, 36, 44, 46, 47, 52, 67, 75, 77, 82, 86, 89, 91, 113, 123, 138, 143, 147, 158, 161, 167, and 175.

This code is used in section 2.

6. ⟨ Set initial values 6 ⟩ ≡
reset(*vpl_file*);

See also sections 22, 26, 28, 30, 32, 45, 49, 68, 80, 84, and 148.

This code is used in section 2.

7. A VPL file is like a PL file with a few extra features, so we can begin to define it by reviewing the definition of PL files. The material in the next few sections is copied from the program PLtoTF.

A PL file is a list of entries of the form

```
(PROPERTYNAME VALUE)
```

where the property name is one of a finite set of names understood by this program, and the value may itself in turn be a property list. The idea is best understood by looking at an example, so let's consider a fragment of the PL file for a hypothetical font.

```
(FAMILY NOVA)
(FACE F MIE)
(CODINGSCHEME ASCII)
(DESIGNSIZE D 10)
(DESIGNUNITS D 18)
(COMMENT A COMMENT IS IGNORED)
(COMMENT (EXCEPT THIS ONE ISN'T))
(COMMENT (ACTUALLY IT IS, EVEN THOUGH
        IT SAYS IT ISN'T))
(FONTDIMEN
    (SLANT R -.25)
    (SPACE D 6)
    (SHRINK D 2)
    (STRETCH D 3)
    (XHEIGHT R 10.55)
    (QUAD D 18)
    )
(LIGTABLE
    (LABEL C f)
    (LIG C f O 200)
    (SKIP D 1)
    (LABEL O 200)
    (LIG C i O 201)
    (KRN O 51 R 1.5)
    (/LIG C ? C f)
    (STOP)
    )
(CHARACTER C f
    (CHARWD D 6)
    (CHARHT R 13.5)
    (CHARIC R 1.5)
    )
```

This example says that the font whose metric information is being described belongs to the hypothetical NOVA family; its face code is medium italic extended; and the characters appear in ASCII code positions. The design size is 10 points, and all other sizes in this PL file are given in units such that 18 units equals the design size. The font is slanted with a slope of $-.25$ (hence the letters actually slant backward—perhaps that is why the family name is NOVA). The normal space between words is 6 units (i.e., one third of the 18-unit design size), with glue that shrinks by 2 units or stretches by 3. The letters for which accents don't need to be raised or lowered are 10.55 units high, and one em equals 18 units.

The example ligature table is a bit trickier. It specifies that the letter f followed by another f is changed to code *´200*, while code *´200* followed by i is changed to *´201*; presumably codes *´200* and *´201* represent the ligatures 'ff' and 'ffi'. Moreover, in both cases f and *´200*, if the following character is the code *´51* (which is a right parenthesis), an additional 1.5 units of space should be inserted before the *´51*. (The 'SKIP D 1' skips over one LIG or KRN command, which in this case is the second LIG; in this way two different ligature/kern programs can come together.) Finally, if either f or *´200* is followed by a question mark, the question mark is replaced by f and the ligature program is started over. (Thus, the character pair 'f?' would actually become the ligature 'ff', and 'ff?' or 'f?f' would become 'fff'. To avoid this restart procedure, the /LIG command could be replaced by /LIG>; then 'f? would become 'ff' and 'f?f' would become 'fff'.)

Character f itself is 6 units wide and 13.5 units tall, in this example. Its depth is zero (since CHARDP is not given), and its italic correction is 1.5 units.

8. The example above illustrates most of the features found in PL files. Note that some property names, like FAMILY or COMMENT, take a string as their value; this string continues until the first unmatched right parenthesis. But most property names, like DESIGNSIZE and SLANT and LABEL, take a number as their value. This number can be expressed in a variety of ways, indicated by a prefixed code; D stands for decimal, H for hexadecimal, O for octal, R for real, C for character, and F for "face." Other property names, like LIG, take two numbers as their value. And still other names, like FONTDIMEN and LIGTABLE and CHARACTER, have more complicated values that involve property lists.

A property name is supposed to be used only in an appropriate property list. For example, CHARWD shouldn't occur on the outer level or within FONTDIMEN.

The individual property-and-value pairs in a property list can appear in any order. For instance, 'SHRINK' precedes 'STRETCH' in the above example, although the TFM file always puts the stretch parameter first. One could even give the information about characters like 'f' before specifying the number of units in the design size, or before specifying the ligature and kerning table. However, the LIGTABLE itself is an exception to this rule; the individual elements of the LIGTABLE property list can be reordered only to a certain extent without changing the meaning of that table.

If property-and-value pairs are omitted, a default value is used. For example, we have already noted that the default for CHARDP is zero. The default for *every* numeric value is, in fact, zero, unless otherwise stated below.

If the same property name is used more than once, VPtoVF will not notice the discrepancy; it simply uses the final value given. Once again, however, the LIGTABLE is an exception to this rule; VPtoVF will complain if there is more than one label for some character. And of course many of the entries in the LIGTABLE property list have the same property name.

9. A VPL file also includes information about how to create each character, by typesetting characters from other fonts and/or by drawing lines, etc. Such information is the value of the 'MAP' property, which can be illustrated as follows:

```
(MAPFONT D 0 (FONTNAME Times-Roman))
(MAPFONT D 1 (FONTNAME Symbol))
(MAPFONT D 2 (FONTNAME cmr10)(FONTAT D 20))
(CHARACTER O 0 (MAP (SELECTFONT D 1)(SETCHAR C G)))
(CHARACTER O 76 (MAP (SETCHAR O 277)))
(CHARACTER D 197 (MAP
    (PUSH)(SETCHAR C A)(POP)
    (MOVEUP R 0.937)(MOVERIGHT R 1.5)(SETCHAR O 312)))
(CHARACTER O 200 (MAP (MOVEDOWN R 2.1)(SETRULE R 1 R 8)))
(CHARACTER O 201 (MAP
    (SPECIAL ps: /SaveGray currentgray def .5 setgray)
    (SELECTFONT D 2)(SETCHAR C A)
    (SPECIAL ps: SaveGray setgray)))
```

(These specifications appear in addition to the conventional PL information. The MAP attribute can be mixed in with other attributes like CHARWD or it can be given separately.)

In this example, the virtual font is composed of characters that can be fabricated from three actual fonts, 'Times-Roman', 'Symbol', and 'cmr10 at 20\u' (where \u is the unit size in this VPL file). Character *´0* is typeset as a 'G' from the symbol font. Character *´76* is typeset as character *´277* from the ordinary Times font. (If no other font is selected, font number 0 is the default. If no MAP attribute is given, the default map is a character of the same number in the default font.)

Character 197 (decimal) is more interesting: First an A is typeset (in the default font Times), and this is enclosed by PUSH and POP so that the original position is restored. Then the accent character *´312* is typeset, after moving up .937 units and right 1.5 units.

To typeset character *´200* in this virtual font, we move down 2.1 units, then typeset a rule that is 1 unit high and 8 units wide.

Finally, to typeset character *´201*, we do something that requires a special ability to interpret PostScript commands; this example sets the PostScript "color" to 50% gray and typesets an 'A' from cmr10 in that color.

In general, the MAP attribute of a virtual character can be any sequence of typesetting commands that might appear in a page of a DVI file. A single character might map into an entire page.

10. But instead of relying on a hypothetical example, let's consider a complete grammar for VPL files, beginning with the (unchanged) grammatical rules for PL files. At the outer level, the following property names are valid in any PL file:

CHECKSUM (four-byte value). The value, which should be a nonnegative integer less than 2^{32}, is used to identify a particular version of a font; it should match the check sum value stored with the font itself. An explicit check sum of zero is used to bypass check sum testing. If no checksum is specified in the VPL file, VPtoVF will compute the checksum that METAFONT would compute from the same data.

DESIGNSIZE (numeric value, default is 10). The value, which should be a real number in the range $1.0 \le x < 2048$, represents the default amount by which all quantities will be scaled if the font is not loaded with an 'at' specification. For example, if one says '\font\A=cmr10 at 15pt' in TeX language, the design size in the TFM file is ignored and effectively replaced by 15 points; but if one simply says '\font\A=cmr10' the stated design size is used. This quantity is always in units of printer's points.

DESIGNUNITS (numeric value, default is 1). The value should be a positive real number; it says how many units equals the design size (or the eventual 'at' size, if the font is being scaled). For example, suppose you have a font that has been digitized with 600 pixels per em, and the design size is one em; then you could say '(DESIGNUNITS R 600)' if you wanted to give all of your measurements in units of pixels.

CODINGSCHEME (string value, default is 'UNSPECIFIED'). The string should not contain parentheses, and its length must be less than 40. It identifies the correspondence between the numeric codes and font characters. (TeX ignores this information, but other software programs make use of it.)

FAMILY (string value, default is 'UNSPECIFIED'). The string should not contain parentheses, and its length must be less than 20. It identifies the name of the family to which this font belongs, e.g., 'HELVETICA'. (TeX ignores this information; but it is needed, for example, when converting DVI files to PRESS files for Xerox equipment.)

FACE (one-byte value). This number, which must lie between 0 and 255 inclusive, is a subsidiary identification of the font within its family. For example, bold italic condensed fonts might have the same family name as light roman extended fonts, differing only in their face byte. (TeX ignores this information; but it is needed, for example, when converting `DVI` files to `PRESS` files for Xerox equipment.)

`SEVENBITSAFEFLAG` (string value, default is '`FALSE`'). The value should start with either '`T`' (true) or '`F`' (false). If true, character codes less than 128 cannot lead to codes of 128 or more via ligatures or charlists or extensible characters. (TeX82 ignores this flag, but older versions of TeX would only accept `TFM` files that were seven-bit safe.) `VPtoVF` computes the correct value of this flag and gives an error message only if a claimed "true" value is incorrect.

`HEADER` (a one-byte value followed by a four-byte value). The one-byte value should be between 18 and a maximum limit that can be raised or lowered depending on the compile-time setting of *max_header_bytes*. The four-byte value goes into the header word whose index is the one-byte value; for example, to set $header[18] \leftarrow 1$, one may write '`(HEADER D 18 O 1)`'. This notation is used for header information that is presently unnamed. (TeX ignores it.)

`FONTDIMEN` (property list value). See below for the names allowed in this property list.

`LIGTABLE` (property list value). See below for the rules about this special kind of property list.

`BOUNDARYCHAR` (one-byte value). If this character appears in a `LIGTABLE` command, it matches "end of word" as well as itself. If no boundary character is given and no `LABEL BOUNDARYCHAR` occurs within `LIGTABLE`, word boundaries will not affect ligatures or kerning.

`CHARACTER`. The value is a one-byte integer followed by a property list. The integer represents the number of a character that is present in the font; the property list of a character is defined below. The default is an empty property list.

11. Numeric property list values can be given in various forms identified by a prefixed letter.

`C` denotes an ASCII character, which should be a standard visible character that is not a parenthesis. The numeric value will therefore be between *´41* and *´176* but not *´50* or *´51*.

`D` denotes an unsigned decimal integer, which must be less than 2^{32}, i.e., at most '`D 4294967295`'.

`F` denotes a three-letter Xerox face code; the admissible codes are `MRR`, `MIR`, `BRR`, `BIR`, `LRR`, `LIR`, `MRC`, `MIC`, `BRC`, `BIC`, `LRC`, `LIC`, `MRE`, `MIE`, `BRE`, `BIE`, `LRE`, and `LIE`, denoting the integers 0 to 17, respectively.

`O` denotes an unsigned octal integer, which must be less than 2^{32}, i.e., at most '`O 37777777777`'.

`H` denotes an unsigned hexadecimal integer, which must be less than 2^{32}, i.e., at most '`H FFFFFFFF`'.

`R` denotes a real number in decimal notation, optionally preceded by a '`+`' or '`-`' sign, and optionally including a decimal point. The absolute value must be less than 2048.

12. The property names allowed in a `FONTDIMEN` property list correspond to various TeX parameters, each of which has a (real) numeric value. All of the parameters except `SLANT` are in design units. The admissible names are `SLANT`, `SPACE`, `STRETCH`, `SHRINK`, `XHEIGHT`, `QUAD`, `EXTRASPACE`, `NUM1`, `NUM2`, `NUM3`, `DENOM1`, `DENOM2`, `SUP1`, `SUP2`, `SUP3`, `SUB1`, `SUB2`, `SUPDROP`, `SUBDROP`, `DELIM1`, `DELIM2`, and `AXISHEIGHT`, for parameters 1 to 22. The alternate names `DEFAULTRULETHICKNESS`, `BIGOPSPACING1`, `BIGOPSPACING2`, `BIGOPSPACING3`, `BIGOPSPACING4`, and `BIGOPSPACING5`, may also be used for parameters 8 to 13.

The notation '`PARAMETER` n' provides another way to specify the nth parameter; for example, '`(PARAMETER D 1 R -.25)`' is another way to specify that the `SLANT` is -0.25. The value of n must be positive and less than *max_param_words*.

13. The elements of a CHARACTER property list can be of six different types.

CHARWD (real value) denotes the character's width in design units.

CHARHT (real value) denotes the character's height in design units.

CHARDP (real value) denotes the character's depth in design units.

CHARIC (real value) denotes the character's italic correction in design units.

NEXTLARGER (one-byte value), specifies the character that follows the present one in a "charlist." The value must be the number of a character in the font, and there must be no infinite cycles of supposedly larger and larger characters.

VARCHAR (property list value), specifies an extensible character. This option and NEXTLARGER are mutually exclusive; i.e., they cannot both be used within the same CHARACTER list.

The elements of a VARCHAR property list are either TOP, MID, BOT or REP; the values are integers, which must be zero or the number or a character in the font. A zero value for TOP, MID, or BOT means that the corresponding piece of the extensible character is absent. A nonzero value, or a REP value of zero, denotes the character code used to make up the top, middle, bottom, or replicated piece of an extensible character.

14. A LIGTABLE property list contains elements of four kinds, specifying a program in a simple command language that TeX uses for ligatures and kerns. If several LIGTABLE lists appear, they are effectively concatenated into a single list.

LABEL (one-byte value) means that the program for the stated character value starts here. The integer must be the number of a character in the font; its CHARACTER property list must not have a NEXTLARGER or VARCHAR field. At least one LIG or KRN step must follow.

LABEL BOUNDARYCHAR means that the program for beginning-of-word ligatures starts here.

LIG (two one-byte values). The instruction '(LIG c r)' means, "If the next character is c, then insert character r and possibly delete the current character and/or c; otherwise go on to the next instruction." Characters r and c must be present in the font. LIG may be immediately preceded or followed by a slash, and then immediately followed by > characters not exceeding the number of slashes. Thus there are eight possible forms:

LIG /LIG /LIG> LIG/ LIG/> /LIG/ /LIG/> /LIG/>>

The slashes specify retention of the left or right original character; the > signs specify passing over the result without further ligature processing.

KRN (a one-byte value and a real value). The instruction '(KRN c r)' means, "If the next character is c, then insert a blank space of width r between the current character character and c; otherwise go on to the next intruction." The value of r, which is in units of the design size, is often negative. Character code c must exist in the font.

STOP (no value). This instruction ends a ligature/kern program. It must follow either a LIG or KRN instruction, not a LABEL or STOP or SKIP.

SKIP (value in the range 0 .. 127). This instruction specifies continuation of a ligature/kern program after the specified number of LIG or KRN has been skipped over. The number of subsequent LIG and KRN instructions must therefore exceed this specified amount.

15. In addition to all these possibilities, the property name COMMENT is allowed in any property list. Such comments are ignored.

16. So that is what PL files hold. In a VPL file additional properties are recognized; two of these are valid on the outermost level:

VTITLE (string value, default is empty). The value will be reproduced at the beginning of the VF file (and printed on the terminal by VFtoVP when it examines that file).

MAPFONT. The value is a nonnegative integer followed by a property list. The integer represents an identifying number for fonts used in MAP attributes. The property list, which identifies the font and relative size, is defined below.

And one additional "virtual property" is valid within a CHARACTER:

MAP. The value is a property list consisting of typesetting commands. Default is the single command SETCHAR c, where c is the current character number.

17. The elements of a MAPFONT property list can be of the following types.

FONTNAME (string value, default is NULL). This is the font's identifying name.

FONTAREA (string value, default is empty). If the font appears in a nonstandard directory, according to local conventions, the directory name is given here. (This is system dependent, just as in DVI files.)

FONTCHECKSUM (four-byte value, default is zero). This value, which should be a nonnegative integer less than 2^{32}, can be used to check that the font being referred to matches the intended font. If nonzero, it should equal the CHECKSUM parameter in that font.

FONTAT (numeric value, default is the DESIGNUNITS of the present virtual font). This value is relative to the design units of the present virtual font, hence it will be scaled when the virtual font is magnified or reduced. It represents the value that will effectively replace the design size of the font being referred to, so that all characters will be scaled appropriately.

FONTDSIZE (numeric value, default is 10). This value is absolute, in units of printer's points. It should equal the DESIGNSIZE parameter in the font being referred to.

If any of the string values contain parentheses, the parentheses must be balanced. Leading blanks are removed from the strings, but trailing blanks are not.

18. Finally, the elements of a MAP property list are an ordered sequence of typesetting commands chosen from among the following:

SELECTFONT (four-byte integer value). The value must be the number of a previously defined MAPFONT. This font (or more precisely, the final font that is mapped to that code number, if two MAPFONT properties happen to specify the same code) will be used in subsequent SETCHAR instructions until overridden by another SELECTFONT. The first-specified MAPFONT is implicitly selected before the first SELECTFONT in every character's map.

SETCHAR (one-byte integer value). There must be a character of this number in the currently selected font. (VPtoVF doesn't check that the character is valid, but VFtoVP does.) That character is typeset at the current position, and the typesetter moves right by the CHARWD in that character's TFM file.

SETRULE (two real values). The first value specifies height, the second specifies width, in design units. If both height and width are positive, a rule is typeset at the current position. Then the typesetter moves right, by the specified width.

MOVERIGHT, MOVELEFT, MOVEUP, MOVEDOWN (real value). The typesetter moves its current position by the number of design units specified.

PUSH The current typesetter position is remembered, to be restored on a subsequent POP.

POP The current typesetter position is reset to where it was on the most recent unmatched PUSH. The PUSH and POP commands in any MAP must be properly nested like balanced parentheses.

SPECIAL (string value). The subsequent characters, starting with the first nonblank and ending just before the first ')' that has no matching '(', are interpreted according to local conventions with the same system-dependent meaning as a 'special' (*xxx*) command in a DVI file.

SPECIALHEX (hexadecimal string value). The subsequent nonblank characters before the next ')' must consist entirely of hexadecimal digits, and they must contain an even number of such digits. Each pair of hex digits specifies a byte, and this string of bytes is treated just as the value of a SPECIAL. (This convention permits arbitrary byte strings to be represented in an ordinary text file.)

19. Virtual font mapping is a recursive process, like macro expansion. Thus, a MAPFONT might specify another virtual font, whose characters are themselves mapped to other fonts. As an example of this possibility, consider the following curious file called recurse.vpl, which defines a virtual font that is self-contained and self-referential:

```
(VTITLE Example of recursion)
(MAPFONT D 0 (FONTNAME recurse)(FONTAT D 2))
(CHARACTER C A (CHARWD D 1)(CHARHT D 1)(MAP (SETRULE D 1 D 1)))
(CHARACTER C B (CHARWD D 2)(CHARHT D 2)(MAP (SETCHAR C A)))
(CHARACTER C C (CHARWD D 4)(CHARHT D 4)(MAP (SETCHAR C B)))
```

The design size is 10 points (the default), hence the character A in font recurse is a 10×10 point black square. Character B is typeset as character A in recurse scaled 2000, hence it is a 20×20 point black square. And character C is typeset as character B in recurse scaled 2000, hence its size is 40×40.

Users are responsible for making sure that infinite recursion doesn't happen.