1. Introduction. The `VFtoVP` utility program converts a virtual font ("`VF`") file and its associated TEX font metric ("`TFM`") file into an equivalent virtual-property-list ("`VPL`") file. It also makes a thorough check of the given files, using algorithms that are essentially the same as those used by `DVI` device drivers and by TEX. Thus if TEX or a `DVI` driver complains that a `TFM` or `VF` file is "bad," this program will pinpoint the source or sources of badness. A `VPL` file output by this program can be edited with a normal text editor, and the result can be converted back to `VF` and `TFM` format using the companion program `VPtoVF`.

`VFtoVP` is an extended version of the program `TFtoPL`, which is part of the standard TEXware library. The idea of a virtual font was inspired by the work of David R. Fuchs who designed a similar set of conventions in 1984 while developing a device driver for ArborText, Inc. He wrote a somewhat similar program called `AMFtoXPL`.

The *banner* string defined here should be changed whenever `VFtoVP` gets modified.

define *banner* ≡ ´This␣is␣VFtoVP,␣Version␣1.3´ { printed when the program starts }

2. This program is written entirely in standard Pascal, except that it occasionally has lower case letters in strings that are output. Such letters can be converted to upper case if necessary. The input is read from *vf_file* and *tfm_file*; the output is written on *vpl_file*. Error messages and other remarks are written on the *output* file, which the user may choose to assign to the terminal if the system permits it.

The term *print* is used instead of *write* when this program writes on the *output* file, so that all such output can be easily deflected.

define *print*(#) ≡ *write*(#)
define *print_ln*(#) ≡ *write_ln*(#)

program *VFtoVP*(*vf_file*, *tfm_file*, *vpl_file*, *output*);
  label ⟨ Labels in the outer block 3 ⟩
  const ⟨ Constants in the outer block 4 ⟩
  type ⟨ Types in the outer block 5 ⟩
  var ⟨ Globals in the outer block 7 ⟩
  procedure *initialize*; { this procedure gets things started properly }
    var *k*: *integer*; { all-purpose index for initialization }
    begin *print_ln*(*banner*);
    ⟨ Set initial values 11 ⟩
    end;

3. If the program has to stop prematurely, it goes to the '*final_end*'.

define *final_end* = 9999 { label for the end of it all }

⟨ Labels in the outer block 3 ⟩ ≡
  final_end;

This code is used in section 2.

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

⟨ Constants in the outer block 4 ⟩ ≡
  tfm_size = 30000; { maximum length of *tfm* data, in bytes }
  vf_size = 10000; { maximum length of *vf* data, in bytes }
  max_fonts = 300; { maximum number of local fonts in the *vf* file }
  lig_size = 5000; { maximum length of *lig_kern* program, in words }
  hash_size = 5003;
    { preferably a prime number, a bit larger than the number of character pairs in lig/kern steps }
  name_length = 50; { a file name shouldn't be longer than this }
  max_stack = 50; { maximum depth of `DVI` stack in character packets }

This code is used in section 2.

5. Here are some macros for common programming idioms.

define $incr(\#) \equiv \# \leftarrow \# + 1$ { increase a variable by unity }
define $decr(\#) \equiv \# \leftarrow \# - 1$ { decrease a variable by unity }
define *do_nothing* $\equiv$ { empty statement }
define $exit = 10$ { go here to leave a procedure }
define $not\_found = 45$ { go here when you've found nothing }
define *return* $\equiv$ **goto** *exit* { terminate a procedure call }
format *return* $\equiv$ *nil*

⟨ Types in the outer block 5 ⟩ $\equiv$
byte $= 0 \mathinner{..} 255$; { unsigned eight-bit quantity }

See also section 22.

This code is used in section 2.

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** *byte*;

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* $\equiv$ 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.

10. Font metric data. The idea behind `TFM` files is that typesetting routines like TeX need a compact way to store the relevant information about several dozen fonts, and computer centers need a compact way to store the relevant information about several hundred fonts. `TFM` files are compact, and most of the information they contain is highly relevant, so they provide a solution to the problem.

The information in a `TFM` file appears in a sequence of 8-bit bytes. Since the number of bytes is always a multiple of 4, we could also regard the file as a sequence of 32-bit words; but TeX uses the byte interpretation, and so does `VFtoVP`. Note that the bytes are considered to be unsigned numbers.

⟨ Globals in the outer block 7 ⟩ +≡
tfm_file: **packed file of** *byte*;

11. On some systems you may have to do something special to read a packed file of bytes. For example, the following code didn't work when it was first tried at Stanford, because packed files have to be opened with a special switch setting on the Pascal that was used.

⟨ Set initial values 11 ⟩ ≡
reset(*tfm_file*); *reset*(*vf_file*);

See also sections 21, 43, 50, 55, 68, and 86.

This code is used in section 2.

12. The first 24 bytes (6 words) of a `TFM` file contain twelve 16-bit integers that give the lengths of the various subsequent portions of the file. These twelve integers are, in order:

$$\begin{aligned}
lf &= \text{length of the entire file, in words;}\\
lh &= \text{length of the header data, in words;}\\
bc &= \text{smallest character code in the font;}\\
ec &= \text{largest character code in the font;}\\
nw &= \text{number of words in the width table;}\\
nh &= \text{number of words in the height table;}\\
nd &= \text{number of words in the depth table;}\\
ni &= \text{number of words in the italic correction table;}\\
nl &= \text{number of words in the lig/kern table;}\\
nk &= \text{number of words in the kern table;}\\
ne &= \text{number of words in the extensible character table;}\\
np &= \text{number of font parameter words.}
\end{aligned}$$

They are all nonnegative and less than 2^{15}. We must have $bc - 1 \le ec \le 255$, $ne \le 256$, and

$$lf = 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np.$$

Note that a font may contain as many as 256 characters (if $bc = 0$ and $ec = 255$), and as few as 0 characters (if $bc = ec + 1$).

Incidentally, when two or more 8-bit bytes are combined to form an integer of 16 or more bits, the most significant bytes appear first in the file. This is called BigEndian order.

⟨ Globals in the outer block 7 ⟩ +≡
lf, *lh*, *bc*, *ec*, *nw*, *nh*, *nd*, *ni*, *nl*, *nk*, *ne*, *np*: 0 .. ´77777; { subfile sizes }

13. The rest of the TFM file may be regarded as a sequence of ten data arrays having the informal specification

$$
\begin{aligned}
\mathit{header} &: \textbf{array } [0 \mathrel{..} lh-1] \textbf{ of } \mathit{stuff}\\
\mathit{char\_info} &: \textbf{array } [bc \mathrel{..} ec] \textbf{ of } \mathit{char\_info\_word}\\
\mathit{width} &: \textbf{array } [0 \mathrel{..} nw-1] \textbf{ of } \mathit{fix\_word}\\
\mathit{height} &: \textbf{array } [0 \mathrel{..} nh-1] \textbf{ of } \mathit{fix\_word}\\
\mathit{depth} &: \textbf{array } [0 \mathrel{..} nd-1] \textbf{ of } \mathit{fix\_word}\\
\mathit{italic} &: \textbf{array } [0 \mathrel{..} ni-1] \textbf{ of } \mathit{fix\_word}\\
\mathit{lig\_kern} &: \textbf{array } [0 \mathrel{..} nl-1] \textbf{ of } \mathit{lig\_kern\_command}\\
\mathit{kern} &: \textbf{array } [0 \mathrel{..} nk-1] \textbf{ of } \mathit{fix\_word}\\
\mathit{exten} &: \textbf{array } [0 \mathrel{..} ne-1] \textbf{ of } \mathit{extensible\_recipe}\\
\mathit{param} &: \textbf{array } [1 \mathrel{..} np] \textbf{ of } \mathit{fix\_word}
\end{aligned}
$$

The most important data type used here is a *fix_word*, which is a 32-bit representation of a binary fraction. A *fix_word* is a signed quantity, with the two's complement of the entire word used to represent negation. Of the 32 bits in a *fix_word*, exactly 12 are to the left of the binary point; thus, the largest *fix_word* value is $2048 - 2^{-20}$, and the smallest is -2048. We will see below, however, that all but one of the *fix_word* values will lie between -16 and $+16$.

14. The first data array is a block of header information, which contains general facts about the font. The header must contain at least two words, and for `TFM` files to be used with Xerox printing software it must contain at least 18 words, allocated as described below. When different kinds of devices need to be interfaced, it may be necessary to add further words to the header block.

header[0] is a 32-bit check sum that TeX will copy into the `DVI` output file whenever it uses the font. Later on when the `DVI` file is printed, possibly on another computer, the actual font that gets used is supposed to have a check sum that agrees with the one in the `TFM` file used by TeX. In this way, users will be warned about potential incompatibilities. (However, if the check sum is zero in either the font file or the `TFM` file, no check is made.) The actual relation between this check sum and the rest of the `TFM` file is not important; the check sum is simply an identification number with the property that incompatible fonts almost always have distinct check sums.

header[1] is a *fix_word* containing the design size of the font, in units of TeX points (7227 TeX points = 254 cm). This number must be at least 1.0; it is fairly arbitrary, but usually the design size is 10.0 for a "10 point" font, i.e., a font that was designed to look best at a 10-point size, whatever that really means. When a TeX user asks for a font '`at` δ `pt`', the effect is to override the design size and replace it by δ, and to multiply the x and y coordinates of the points in the font image by a factor of δ divided by the design size. *All other dimensions in the* `TFM` *file are fix_word numbers in design-size units.* Thus, for example, the value of *param*[6], one `em` or `\quad`, is often the *fix_word* value $2^{20} = 1.0$, since many fonts have a design size equal to one em. The other dimensions must be less than 16 design-size units in absolute value; thus, *header*[1] and *param*[1] are the only *fix_word* entries in the whole `TFM` file whose first byte might be something besides 0 or 255.

header[2 .. 11], if present, contains 40 bytes that identify the character coding scheme. The first byte, which must be between 0 and 39, is the number of subsequent ASCII bytes actually relevant in this string, which is intended to specify what character-code-to-symbol convention is present in the font. Examples are `ASCII` for standard ASCII, `TeX text` for fonts like `cmr10` and `cmti9`, `TeX math extension` for `cmex10`, `XEROX text` for Xerox fonts, `GRAPHIC` for special-purpose non-alphabetic fonts, `UNSPECIFIED` for the default case when there is no information. Parentheses should not appear in this name. (Such a string is said to be in BCPL format.)

header[12 .. 16], if present, contains 20 bytes that name the font family (e.g., `CMR` or `HELVETICA`), in BCPL format. This field is also known as the "font identifier."

header[17], if present, contains a first byte called the *seven_bit_safe_flag*, then two bytes that are ignored, and a fourth byte called the *face*. If the value of the fourth byte is less than 18, it has the following interpretation as a "weight, slope, and expansion": Add 0 or 2 or 4 (for medium or bold or light) to 0 or 1 (for roman or italic) to 0 or 6 or 12 (for regular or condensed or extended). For example, 13 is 0+1+12, so it represents medium italic extended. A three-letter code (e.g., `MIE`) can be used for such *face* data.

header[18 .. whatever] might also be present; the individual words are simply called *header*[18], *header*[19], etc., at the moment.

15. Next comes the *char_info* array, which contains one *char_info_word* per character. Each *char_info_word* contains six fields packed into four bytes as follows.

first byte: *width_index* (8 bits)
second byte: *height_index* (4 bits) times 16, plus *depth_index* (4 bits)
third byte: *italic_index* (6 bits) times 4, plus *tag* (2 bits)
fourth byte: *remainder* (8 bits)

The actual width of a character is *width*[*width_index*], in design-size units; this is a device for compressing information, since many characters have the same width. Since it is quite common for many characters to have the same height, depth, or italic correction, the `TFM` format imposes a limit of 16 different heights, 16 different depths, and 64 different italic corrections.

Incidentally, the relation $width[0] = height[0] = depth[0] = italic[0] = 0$ should always hold, so that an index of zero implies a value of zero. The *width_index* should never be zero unless the character does not exist in the font, since a character is valid if and only if it lies between *bc* and *ec* and has a nonzero *width_index*.

16. The *tag* field in a *char_info_word* has four values that explain how to interpret the *remainder* field.

$tag = 0$ (*no_tag*) means that *remainder* is unused.
$tag = 1$ (*lig_tag*) means that this character has a ligature/kerning program starting at *lig_kern*[*remainder*].
$tag = 2$ (*list_tag*) means that this character is part of a chain of characters of ascending sizes, and not the largest in the chain. The *remainder* field gives the character code of the next larger character.
$tag = 3$ (*ext_tag*) means that this character code represents an extensible character, i.e., a character that is built up of smaller pieces so that it can be made arbitrarily large. The pieces are specified in *exten*[*remainder*].

define $no\_tag = 0$ { vanilla character }
define $lig\_tag = 1$ { character has a ligature/kerning program }
define $list\_tag = 2$ { character has a successor in a charlist }
define $ext\_tag = 3$ { character is extensible }

17. The *lig_kern* array contains instructions in a simple programming language that explains what to do for special letter pairs. Each word is a *lig_kern_command* of four bytes.

first byte: *skip_byte*, indicates that this is the final program step if the byte is 128 or more, otherwise the next step is obtained by skipping this number of intervening steps.

second byte: *next_char*, "if *next_char* follows the current character, then perform the operation and stop, otherwise continue."

third byte: *op_byte*, indicates a ligature step if less than 128, a kern step otherwise.

fourth byte: *remainder*.

In a kern step, an additional space equal to $kern[256 * (op\_byte - 128) + remainder]$ is inserted between the current character and *next_char*. This amount is often negative, so that the characters are brought closer together by kerning; but it might be positive.

There are eight kinds of ligature steps, having *op_byte* codes $4a+2b+c$ where $0 \le a \le b+c$ and $0 \le b, c \le 1$. The character whose code is *remainder* is inserted between the current character and *next_char*; then the current character is deleted if $b = 0$, and *next_char* is deleted if $c = 0$; then we pass over a characters to reach the next current character (which may have a ligature/kerning program of its own).

Notice that if $a = 0$ and $b = 1$, the current character is unchanged; if $a = b$ and $c = 1$, the current character is changed but the next character is unchanged. `VFtoVP` will check to see that infinite loops are avoided.

If the very first instruction of the *lig_kern* array has $skip\_byte = 255$, the *next_char* byte is the so-called right boundary character of this font; the value of *next_char* need not lie between *bc* and *ec*. If the very last instruction of the *lig_kern* array has $skip\_byte = 255$, there is a special ligature/kerning program for a left boundary character, beginning at location $256 * op\_byte + remainder$. The interpretation is that TeX puts implicit boundary characters before and after each consecutive string of characters from the same font. These implicit characters do not appear in the output, but they can affect ligatures and kerning.

If the very first instruction of a character's *lig_kern* program has $skip\_byte > 128$, the program actually begins in location $256 * op\_byte + remainder$. This feature allows access to large *lig_kern* arrays, because the first instruction must otherwise appear in a location ≤ 255.

Any instruction with $skip\_byte > 128$ in the *lig_kern* array must have $256 * op\_byte + remainder < nl$. If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature command is performed.

define $stop\_flag = 128$ { value indicating '`STOP`' in a lig/kern program }
define $kern\_flag = 128$ { op code for a kern step }

18. Extensible characters are specified by an *extensible_recipe*, which consists of four bytes called *top*, *mid*, *bot*, and *rep* (in this order). These bytes are the character codes of individual pieces used to build up a large symbol. If *top*, *mid*, or *bot* are zero, they are not present in the built-up result. For example, an extensible vertical line is like an extensible bracket, except that the top and bottom pieces are missing.

19. The final portion of a TFM file is the *param* array, which is another sequence of *fix_word* values.

param[1] = *slant* is the amount of italic slant, which is used to help position accents. For example, *slant* = .25 means that when you go up one unit, you also go .25 units to the right. The *slant* is a pure number; it's the only *fix_word* other than the design size itself that is not scaled by the design size.

param[2] = *space* is the normal spacing between words in text. Note that character "␣" in the font need not have anything to do with blank spaces.

param[3] = *space_stretch* is the amount of glue stretching between words.

param[4] = *space_shrink* is the amount of glue shrinking between words.

param[5] = *x_height* is the height of letters for which accents don't have to be raised or lowered.

param[6] = *quad* is the size of one em in the font.

param[7] = *extra_space* is the amount added to *param*[2] at the ends of sentences.

When the character coding scheme is `TeX math symbols`, the font is supposed to have 15 additional parameters called *num1*, *num2*, *num3*, *denom1*, *denom2*, *sup1*, *sup2*, *sup3*, *sub1*, *sub2*, *supdrop*, *subdrop*, *delim1*, *delim2*, and *axis_height*, respectively. When the character coding scheme is `TeX math extension`, the font is supposed to have six additional parameters called *default_rule_thickness* and *big_op_spacing1* through *big_op_spacing5*.

20. So that is what TFM files hold. The next question is, "What about VPL files?" A complete answer to that question appears in the documentation of the companion program, VPtoVF, so it will not be repeated here. Suffice it to say that a VPL file is an ordinary Pascal text file, and that the output of VFtoVP uses only a subset of the possible constructions that might appear in a VPL file. Furthermore, hardly anybody really wants to look at the formal definition of VPL format, because it is almost self-explanatory when you see an example or two.

⟨ Globals in the outer block 7 ⟩ +≡
vpl_file: *text*;

21. ⟨ Set initial values 11 ⟩ +≡
rewrite(*vpl_file*);

22. Unpacking the TFM file. The first thing VFtoVP does is read the entire *tfm_file* into an array of bytes, *tfm*[0 . . (4 ∗ *lf* − 1)].

```
⟨ Types in the outer block 5 ⟩ +≡
  index = 0 . . tfm_size;   { address of a byte in tfm }
```

23.
```
⟨ Globals in the outer block 7 ⟩ +≡
tfm: array [−1000 . . tfm_size] of byte;   { the TFM input data all goes here }
        { the negative addresses avoid range checks for invalid characters }
```

24. The input may, of course, be all screwed up and not a TFM file at all. So we begin cautiously.

```
define abort(#) ≡
          begin print_ln(#);
          print_ln('Sorry, but I can''t go on; are you sure this is a TFM?'); goto final_end;
          end
⟨ Read the whole TFM file 24 ⟩ ≡
  read(tfm_file, tfm[0]);
  if tfm[0] > 127 then abort('The first byte of the input file exceeds 127!');
  if eof(tfm_file) then abort('The input file is only one byte long!');
  read(tfm_file, tfm[1]); lf ← tfm[0] ∗ '400 + tfm[1];
  if lf = 0 then abort('The file claims to have length zero, but that''s impossible!');
  if 4 ∗ lf − 1 > tfm_size then abort('The file is bigger than I can handle!');
  for tfm_ptr ← 2 to 4 ∗ lf − 1 do
    begin if eof(tfm_file) then abort('The file has fewer bytes than it claims!');
    read(tfm_file, tfm[tfm_ptr]);
    end;
  if ¬eof(tfm_file) then
    begin print_ln('There''s some extra junk at the end of the TFM file,');
    print_ln('but I''ll proceed as if it weren''t there.');
    end
```

This code is used in section 131.

25. After the file has been read successfully, we look at the subfile sizes to see if they check out.

```
define eval_two_bytes(#) ≡
        begin if tfm[tfm_ptr] > 127 then abort('One␣of␣the␣subfile␣sizes␣is␣negative!');
        # ← tfm[tfm_ptr] * '400 + tfm[tfm_ptr + 1]; tfm_ptr ← tfm_ptr + 2;
        end
```

⟨ Set subfile sizes *lh*, *bc*, ..., *np* 25 ⟩ ≡

```
begin tfm_ptr ← 2;
eval_two_bytes(lh); eval_two_bytes(bc); eval_two_bytes(ec); eval_two_bytes(nw); eval_two_bytes(nh);
eval_two_bytes(nd); eval_two_bytes(ni); eval_two_bytes(nl); eval_two_bytes(nk); eval_two_bytes(ne);
eval_two_bytes(np);
if lh < 2 then abort('The␣header␣length␣is␣only␣', lh : 1, '!');
if nl > 4 * lig_size then abort('The␣lig/kern␣program␣is␣longer␣than␣I␣can␣handle!');
if (bc > ec + 1) ∨ (ec > 255) then
  abort('The␣character␣code␣range␣', bc : 1, '..', ec : 1, 'is␣illegal!');
if (nw = 0) ∨ (nh = 0) ∨ (nd = 0) ∨ (ni = 0) then
  abort('Incomplete␣subfiles␣for␣character␣dimensions!');
if ne > 256 then abort('There␣are␣', ne : 1, '␣extensible␣recipes!');
if lf ≠ 6 + lh + (ec − bc + 1) + nw + nh + nd + ni + nl + nk + ne + np then
  abort('Subfile␣sizes␣don''t␣add␣up␣to␣the␣stated␣total!');
end
```

This code is used in section 131.

26. Once the input data successfully passes these basic checks, VFtoVP believes that it is a TFM file, and the conversion to VPL format will take place. Access to the various subfiles is facilitated by computing the following base addresses. For example, the *char_info* for character c will start in location $4 * (char\_base + c)$ of the *tfm* array.

⟨ Globals in the outer block 7 ⟩ +≡

```
char_base, width_base, height_base, depth_base, italic_base, lig_kern_base, kern_base, exten_base, param_base:
        integer; { base addresses for the subfiles }
```

27. ⟨ Compute the base addresses 27 ⟩ ≡

```
begin char_base ← 6 + lh − bc; width_base ← char_base + ec + 1; height_base ← width_base + nw;
depth_base ← height_base + nh; italic_base ← depth_base + nd; lig_kern_base ← italic_base + ni;
kern_base ← lig_kern_base + nl; exten_base ← kern_base + nk; param_base ← exten_base + ne − 1;
end
```

This code is used in section 131.

28. Of course we want to define macros that suppress the detail of how the font information is actually encoded. Each word will be referred to by the *tfm* index of its first byte. For example, if c is a character code between bc and ec, then $tfm[char\_info(c)]$ will be the first byte of its *char_info*, i.e., the *width_index*; furthermore $width(c)$ will point to the *fix_word* for c's width.

define $check\_sum = 24$
define $design\_size = check\_sum + 4$
define $scheme = design\_size + 4$
define $family = scheme + 40$
define $random\_word = family + 20$
define $char\_info(\#) \equiv 4 * (char\_base + \#)$
define $width\_index(\#) \equiv tfm[char\_info(\#)]$
define $nonexistent(\#) \equiv ((\# < bc) \lor (\# > ec) \lor (width\_index(\#) = 0))$
define $height\_index(\#) \equiv (tfm[char\_info(\#) + 1]\ \mathbf{div}\ 16)$
define $depth\_index(\#) \equiv (tfm[char\_info(\#) + 1]\ \mathbf{mod}\ 16)$
define $italic\_index(\#) \equiv (tfm[char\_info(\#) + 2]\ \mathbf{div}\ 4)$
define $tag(\#) \equiv (tfm[char\_info(\#) + 2]\ \mathbf{mod}\ 4)$
define $reset\_tag(\#) \equiv tfm[char\_info(\#) + 2] \leftarrow 4 * italic\_index(\#) + no\_tag$
define $remainder(\#) \equiv tfm[char\_info(\#) + 3]$
define $width(\#) \equiv 4 * (width\_base + width\_index(\#))$
define $height(\#) \equiv 4 * (height\_base + height\_index(\#))$
define $depth(\#) \equiv 4 * (depth\_base + depth\_index(\#))$
define $italic(\#) \equiv 4 * (italic\_base + italic\_index(\#))$
define $exten(\#) \equiv 4 * (exten\_base + remainder(\#))$
define $lig\_step(\#) \equiv 4 * (lig\_kern\_base + (\#))$
define $kern(\#) \equiv 4 * (kern\_base + \#)$ { here `#` is an index, not a character }
define $param(\#) \equiv 4 * (param\_base + \#)$ { likewise }

29. One of the things we would like to do is take cognizance of fonts whose character coding scheme is `TeX math symbols` or `TeX math extension`; we will set the *font_type* variable to one of the three choices *vanilla*, *mathsy*, or *mathex*.

define $vanilla = 0$ { not a special scheme }
define $mathsy = 1$ { `TeX math symbols` scheme }
define $mathex = 2$ { `TeX math extension` scheme }

⟨ Globals in the outer block 7 ⟩ +≡
font_type: *vanilla* .. *mathex*; { is this font special? }

30. Unpacking the VF file. Once the `TFM` file has been brought into memory, `VFtoVP` completes the input phase by reading the `VF` information into another array of bytes. In this case we don't store all the data; we check the redundant bytes for consistency with their `TFM` counterparts, and we partially decode the packets.

⟨ Globals in the outer block 7 ⟩ +≡
vf: **array** [0 . . *vf_size*] **of** *byte*; { the `VF` input data goes here }
font_number: **array** [0 . . *max_fonts*] **of** *integer*; { local font numbers }
font_start, *font_chars*: **array** [0 . . *max_fonts*] **of** 0 . . *vf_size*; { font info }
font_ptr: 0 . . *max_fonts*; { number of local fonts }
packet_start, *packet_end*: **array** [*byte*] **of** 0 . . *vf_size*; { character packet boundaries }
packet_found: *boolean*; { at least one packet has appeared }
temp_byte: *byte*; *count*: *integer*; { registers for simple calculations }
real_dsize: *real*; { the design size, converted to floating point }
pl: *integer*; { packet length }
vf_ptr: 0 . . *vf_size*; { first unused location in *vf* }
vf_count: *integer*; { number of bytes read from *vf_file* }

31. Again we cautiously verify that we've been given decent data.

define *read_vf*(#) ≡ *read*(*vf_file*, #)
define *vf_abort*(#) ≡
  begin *print_ln*(#); *print_ln*(´Sorry,␣but␣I␣can´´t␣go␣on;␣are␣you␣sure␣this␣is␣a␣VF?´);
  goto *final_end*;
  end

⟨ Read the whole `VF` file 31 ⟩ ≡
read_vf(*temp_byte*);
if *temp_byte* ≠ *pre* **then** *vf_abort*(´The␣first␣byte␣isn´´t␣`pre´´!´);
⟨ Read the preamble command 32 ⟩;
⟨ Read and store the font definitions and character packets 33 ⟩;
⟨ Read and verify the postamble 34 ⟩

This code is used in section 131.

32. **define** *vf_store*(#) $\equiv$

```
    if vf_ptr + # ≥ vf_size then vf_abort('The␣file␣is␣bigger␣than␣I␣can␣handle!');
    for k ← vf_ptr to vf_ptr + # − 1 do
      begin if eof(vf_file) then vf_abort('The␣file␣ended␣prematurely!');
      read_vf(vf[k]);
      end;
    vf_count ← vf_count + #; vf_ptr ← vf_ptr + #
```

⟨ Read the preamble command 32 ⟩ $\equiv$

```
if eof(vf_file) then vf_abort('The␣input␣file␣is␣only␣one␣byte␣long!');
read_vf(temp_byte);
if temp_byte ≠ id_byte then vf_abort('Wrong␣VF␣version␣number␣in␣second␣byte!');
if eof(vf_file) then vf_abort('The␣input␣file␣is␣only␣two␣bytes␣long!');
read_vf(temp_byte);  { read the length of introductory comment }
vf_count ← 11; vf_ptr ← 0; vf_store(temp_byte);
for k ← 0 to vf_ptr − 1 do print(xchr[vf[k]]);
print_ln('␣'); count ← 0;
for k ← 0 to 7 do
  begin if eof(vf_file) then vf_abort('The␣file␣ended␣prematurely!');
  read_vf(temp_byte);
  if temp_byte = tfm[check_sum + k] then incr(count);
  end;
real_dsize ← (((tfm[design_size] * 256 + tfm[design_size + 1]) * 256 + tfm[design_size + 2]) * 256 +
    tfm[design_size + 3])/'4000000;
if count ≠ 8 then
  begin print_ln('Check␣sum␣and/or␣design␣size␣mismatch.');
  print_ln('Data␣from␣TFM␣file␣will␣be␣assumed␣correct.');
  end
```

This code is used in section 31.

33. ⟨ Read and store the font definitions and character packets 33 ⟩ $\equiv$

```
for k ← 0 to 255 do packet_start[k] ← vf_size;
font_ptr ← 0; packet_found ← false; font_start[0] ← vf_ptr;
repeat if eof(vf_file) then
    begin print_ln('File␣ended␣without␣a␣postamble!'); temp_byte ← post;
    end
  else begin read_vf(temp_byte); incr(vf_count);
    if temp_byte ≠ post then
      if temp_byte > long_char then ⟨ Read and store a font definition 35 ⟩
      else ⟨ Read and store a character packet 46 ⟩;
    end;
until temp_byte = post
```

This code is used in section 31.

34. ⟨ Read and verify the postamble 34 ⟩ ≡

```
while (temp_byte = post) ∧ ¬eof(vf_file) do
  begin read_vf(temp_byte); incr(vf_count);
  end;
if ¬eof(vf_file) then
  begin print_ln('There''s␣some␣extra␣junk␣at␣the␣end␣of␣the␣VF␣file.');
  print_ln('I''ll␣proceed␣as␣if␣it␣weren''t␣there.');
  end;
if vf_count mod 4 ≠ 0 then print_ln('VF␣data␣not␣a␣multiple␣of␣4␣bytes')
```

This code is used in section 31.

35. ⟨ Read and store a font definition 35 ⟩ ≡

```
begin if packet_found ∨ (temp_byte ≥ pre) then
  vf_abort('Illegal␣byte␣', temp_byte : 1, '␣at␣beginning␣of␣character␣packet!');
font_number[font_ptr] ← vf_read(temp_byte − fnt_def1 + 1);
if font_ptr = max_fonts then vf_abort('I␣can''t␣handle␣that␣many␣fonts!');
vf_store(14); { c[4] s[4] d[4] a[1] l[1] }
if vf[vf_ptr − 10] > 0 then { s is negative or exceeds 2^24 − 1 }
  vf_abort('Mapped␣font␣size␣is␣too␣big!');
a ← vf[vf_ptr − 2]; l ← vf[vf_ptr − 1]; vf_store(a + l); { n[a + l] }
⟨ Print the name of the local font 36 ⟩;
⟨ Read the local font's TFM file and record the characters it contains 39 ⟩;
incr(font_ptr); font_start[font_ptr] ← vf_ptr;
end
```

This code is used in section 33.

36. The font area may need to be separated from the font name on some systems. Here we simply reproduce the font area and font name (with no space or punctuation between them).

⟨ Print the name of the local font 36 ⟩ ≡

```
print('MAPFONT␣', font_ptr : 1, ':␣');
for k ← font_start[font_ptr] + 14 to vf_ptr − 1 do print(xchr[vf[k]]);
k ← font_start[font_ptr] + 5;
print_ln('␣at␣', (((vf[k] * 256 + vf[k + 1]) * 256 + vf[k + 2])/'4000000) * real_dsize : 2 : 2, 'pt')
```

This code is used in section 35.

37. Now we must read in another TFM file. But this time we needn't be so careful, because we merely want to discover which characters are present. The next few sections of the program are copied pretty much verbatim from DVItype, so that system-dependent modifications can be copied from existing software.

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

⟨ Globals in the outer block 7 ⟩ +≡

```
a: integer; { length of the area/directory spec }
l: integer; { length of the font name proper }
cur_name: packed array [1 .. name_length] of char; { external name, with no lower case letters }
b0, b1, b2, b3: byte; { four bytes input at once }
font_lh: 0 .. '77777; { header length of current local font }
font_bc, font_ec: 0 .. '77777; { character range of current local font }
```

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

```
define read_tfm(#) ≡
        if eof(tfm_file) then # ← 0 else read(tfm_file, #)
procedure read_tfm_word;
  begin read_tfm(b0); read_tfm(b1); read_tfm(b2); read_tfm(b3);
  end;
```

39. We use the *vf* array to store a list of all valid characters in the local font, beginning at location *font_chars*[*f*].

```
⟨Read the local font's TFM file and record the characters it contains 39⟩ ≡
  font_chars[font_ptr] ← vf_ptr; ⟨Move font name into the cur_name string 44⟩;
  reset(tfm_file, cur_name);
  if eof(tfm_file) then print_ln('---not␣loaded,␣TFM␣file␣can''t␣be␣opened!')
  else begin font_bc ← 0; font_ec ← 256;  {will cause error if not modified soon}
    read_tfm_word;
    if b2 < 128 then
      begin font_lh ← b2 * 256 + b3; read_tfm_word;
      if (b0 < 128) ∧ (b2 < 128) then
        begin font_bc ← b0 * 256 + b1; font_ec ← b2 * 256 + b3;
        end;
      end;
    if font_bc ≤ font_ec then
      if font_ec > 255 then print_ln('---not␣loaded,␣bad␣TFM␣file!')
      else begin for k ← 0 to 3 + font_lh do
            begin read_tfm_word;
            if k = 4 then ⟨Check the check sum 40⟩;
            if k = 5 then ⟨Check the design size 41⟩;
            end;
          for k ← font_bc to font_ec do
            begin read_tfm_word;
            if b0 > 0 then  {character k exists in the font}
              begin vf[vf_ptr] ← k; incr(vf_ptr);
              if vf_ptr = vf_size then vf_abort('I''m␣out␣of␣VF␣memory!');
              end;
            end;
          end;
    if eof(tfm_file) then print_ln('---trouble␣is␣brewing,␣TFM␣file␣ended␣too␣soon!');
    end;
  incr(vf_ptr)  {leave space for character search later}
```

This code is used in section 35.

40.
```
⟨Check the check sum 40⟩ ≡
  if b0 + b1 + b2 + b3 > 0 then
    if (b0 ≠ vf[font_start[font_ptr]]) ∨ (b1 ≠ vf[font_start[font_ptr] + 1]) ∨
          (b2 ≠ vf[font_start[font_ptr] + 2]) ∨ (b3 ≠ vf[font_start[font_ptr] + 3]) then
      begin print_ln('Check␣sum␣in␣VF␣file␣being␣replaced␣by␣TFM␣check␣sum');
      vf[font_start[font_ptr]] ← b0; vf[font_start[font_ptr] + 1] ← b1; vf[font_start[font_ptr] + 2] ← b2;
      vf[font_start[font_ptr] + 3] ← b3;
      end
```

This code is used in section 39.

41. $\langle$ Check the design size 41 $\rangle \equiv$
if $(b0 \neq vf[font\_start[font\_ptr] + 8]) \lor (b1 \neq vf[font\_start[font\_ptr] + 9]) \lor$
$(b2 \neq vf[font\_start[font\_ptr] + 10]) \lor (b3 \neq vf[font\_start[font\_ptr] + 11])$ **then**
begin *print_ln*(`´Design␣size␣in␣VF␣file␣being␣replaced␣by␣TFM␣design␣size´`);
$vf[font\_start[font\_ptr] + 8] \leftarrow b0$; $vf[font\_start[font\_ptr] + 9] \leftarrow b1$; $vf[font\_start[font\_ptr] + 10] \leftarrow b2$;
$vf[font\_start[font\_ptr] + 11] \leftarrow b3$;
end

This code is used in section 39.

42. If no font directory has been specified, `DVI`-reading software is supposed to use the default font directory, which is a system-dependent place where the standard fonts are kept. The string variable *default_directory* contains the name of this area.

define *default_directory_name* $\equiv$ `´TeXfonts:´` { change this to the correct name }
define *default_directory_name_length* $= 9$ { change this to the correct length }

$\langle$ Globals in the outer block 7 $\rangle +\equiv$
default_directory: **packed array** [1 .. *default_directory_name_length*] **of** *char*;

43. $\langle$ Set initial values 11 $\rangle +\equiv$
default_directory $\leftarrow$ *default_directory_name*;

44. The string *cur_name* is supposed to be set to the external name of the `TFM` file for the current font. This usually means that we need to prepend the name of the default directory, and to append the suffix '`.TFM`'. Furthermore, we change lower case letters to upper case, since *cur_name* is a Pascal string.

$\langle$ Move font name into the *cur_name* string 44 $\rangle \equiv$
for $k \leftarrow 1$ **to** *name_length* **do** $cur\_name[k] \leftarrow$ `´␣´`;
if $a = 0$ **then**
begin for $k \leftarrow 1$ **to** *default_directory_name_length* **do** $cur\_name[k] \leftarrow default\_directory[k]$;
$r \leftarrow default\_directory\_name\_length$;
end
else $r \leftarrow 0$;
for $k \leftarrow font\_start[font\_ptr] + 14$ **to** $vf\_ptr - 1$ **do**
begin *incr*(r);
if $r + 4 > name\_length$ **then** *vf_abort*(`´Font␣name␣too␣long␣for␣me!´`);
if $(vf[k] \geq$ `"a"`$) \land (vf[k] \leq$ `"z"`$)$ **then** $cur\_name[r] \leftarrow xchr[vf[k] - ´40]$
else $cur\_name[r] \leftarrow xchr[vf[k]]$;
end;
$cur\_name[r + 1] \leftarrow$ `´.´`; $cur\_name[r + 2] \leftarrow$ `´T´`; $cur\_name[r + 3] \leftarrow$ `´F´`; $cur\_name[r + 4] \leftarrow$ `´M´`

This code is used in section 39.

45. It's convenient to have a subroutine that reads a k-byte number from *vf_file*.

```
define get_vf(#) ≡
        if eof(vf_file) then # ← 0 else read_vf(#)
function vf_read(k : integer): integer;  { actually 1 ≤ k ≤ 4 }
  var b: byte;  { input byte }
    a: integer;  { accumulator }
  begin vf_count ← vf_count + k; get_vf(b); a ← b;
  if k = 4 then
    if b ≥ 128 then a ← a − 256;  { 4-byte numbers are signed }
  while k > 1 do
    begin get_vf(b); a ← 256 ∗ a + b; decr(k);
    end;
  vf_read ← a;
  end;
```

46. The VF format supports arbitrary 4-byte character codes, but VPL format presently does not. Therefore we give up if the character code is not between 0 and 255.

After more experience is gained with present-day VPL files, the best way to extend them to arbitrary character codes will become clear; the extensions to VFtoVP and VPtoVF should not be difficult.

```
⟨ Read and store a character packet 46 ⟩ ≡
  begin if temp_byte = long_char then
    begin pl ← vf_read(4); c ← vf_read(4); count ← vf_read(4);  { pl[4] cc[4] tfm[4] }
    end
  else begin pl ← temp_byte; c ← vf_read(1); count ← vf_read(3);  { pl[1] cc[1] tfm[3] }
    end;
  if nonexistent(c) then vf_abort(´Character␣´, c : 1, ´␣does␣not␣exist!´);
  if packet_start[c] < vf_size then print_ln(´Discarding␣earlier␣packet␣for␣character␣´, c : 1);
  if count ≠ tfm_width(c) then print_ln(´Incorrect␣TFM␣width␣for␣character␣´, c : 1, ´␣in␣VF␣file´);
  if pl < 0 then vf_abort(´Negative␣packet␣length!´);
  packet_start[c] ← vf_ptr; vf_store(pl); packet_end[c] ← vf_ptr − 1; packet_found ← true;
  end
```

This code is used in section 33.

47. The preceding code requires a simple subroutine that evaluates TFM data.

```
function tfm_width(c : byte): integer;
  var a: integer;  { accumulator }
    k: index;  { index into tfm }
  begin k ← width(c);  { we assume that character c exists }
  a ← tfm[k];
  if a ≥ 128 then a ← a − 256;
  tfm_width ← ((256 ∗ a + tfm[k + 1]) ∗ 256 + tfm[k + 2]) ∗ 256 + tfm[k + 3];
  end;
```

48. Basic output subroutines. Let us now define some procedures that will reduce the rest of `VFtoVP`'s work to a triviality.

First of all, it is convenient to have an abbreviation for output to the `VPL` file:

define *out*(#) ≡ *write*(*vpl_file*, #)

49. In order to stick to standard Pascal, we use an *xchr* array to do appropriate conversion of ASCII codes. Three other little strings are used to produce *face* codes like `MIE`.

```
⟨Globals in the outer block 7⟩ +≡
ASCII_04, ASCII_10, ASCII_14: packed array [1 .. 32] of char;
        { strings for output in the user's external character set }
xchr: packed array [0 .. 255] of char;
MBL_string, RI_string, RCE_string: packed array [1 .. 3] of char;
        { handy string constants for face codes }
```

50. ⟨Set initial values 11⟩ +≡

```
ASCII_04 ← '␣!"#$%&''()*+,-./0123456789:;<=>?';
ASCII_10 ← '@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_';
ASCII_14 ← '`abcdefghijklmnopqrstuvwxyz{|}~?';
for k ← 0 to 255 do xchr[k] ← '?';
for k ← 0 to '37 do
  begin xchr[k + '40] ← ASCII_04[k + 1]; xchr[k + '100] ← ASCII_10[k + 1];
  xchr[k + '140] ← ASCII_14[k + 1];
  end;
MBL_string ← 'MBL'; RI_string ← 'RI␣'; RCE_string ← 'RCE';
```

51. The array *dig* will hold a sequence of digits to be output.

```
⟨Globals in the outer block 7⟩ +≡
dig: array [0 .. 11] of 0 .. 9;
```

52. Here, in fact, are two procedures that output $dig[j-1] \ldots dig[0]$, given $j > 0$.

```
procedure out_digs(j : integer); { outputs j digits }
  begin repeat decr(j); out(dig[j] : 1);
  until j = 0;
  end;
procedure print_digs(j : integer); { prints j digits }
  begin repeat decr(j); print(dig[j] : 1);
  until j = 0;
  end;
```

53. The *print_octal* procedure indicates how *print_digs* can be used. Since this procedure is used only to print character codes, it always produces three digits.

```
procedure print_octal(c : byte); { prints octal value of c }
  var j: 0 .. 2; { index into dig }
  begin print(''''); { an apostrophe indicates the octal notation }
  for j ← 0 to 2 do
    begin dig[j] ← c mod 8; c ← c div 8;
    end;
  print_digs(3);
  end;
```

54. A VPL file has nested parentheses, and we want to format the output so that its structure is clear. The *level* variable keeps track of the depth of nesting.

⟨ Globals in the outer block 7 ⟩ +≡
level: 0 . . 5;

55. ⟨ Set initial values 11 ⟩ +≡
level ← 0;

56. Three simple procedures suffice to produce the desired structure in the output.

```
procedure out_ln; { finishes one line, indents the next }
  var l: 0 . . 5;
  begin write_ln(vpl_file);
  for l ← 1 to level do out(´␣␣␣´);
  end;
procedure left; { outputs a left parenthesis }
  begin incr(level); out(´(´);
  end;
procedure right; { outputs a right parenthesis and finishes a line }
  begin decr(level); out(´)´); out_ln;
  end;
```

57. The value associated with a property can be output in a variety of ways. For example, we might want to output a BCPL string that begins in *tfm*[*k*]:

```
procedure out_BCPL(k : index); { outputs a string, preceded by a blank space }
  var l: 0 . . 39; { the number of bytes remaining }
  begin out(´␣´); l ← tfm[k];
  while l > 0 do
    begin incr(k); decr(l); out(xchr[tfm[k]]);
    end;
  end;
```

58. The property value might also be a sequence of l bytes, beginning in *tfm*[*k*], that we would like to output in octal notation. The following procedure assumes that $l \le 4$, but larger values of l could be handled easily by enlarging the *dig* array and increasing the upper bounds on b and j.

```
procedure out_octal(k, l : index); { outputs l bytes in octal }
  var a: 0 . . ´1777; { accumulator for bits not yet output }
    b: 0 . . 32; { the number of significant bits in a }
    j: 0 . . 11; { the number of digits of output }
  begin out(´␣O␣´); { specify octal format }
  a ← 0; b ← 0; j ← 0;
  while l > 0 do ⟨ Reduce l by one, preserving the invariants 59 ⟩;
  while (a > 0) ∨ (j = 0) do
    begin dig[j] ← a mod 8; a ← a div 8; incr(j);
    end;
  out_digs(j);
  end;
```

59. ⟨Reduce l by one, preserving the invariants 59⟩ ≡

```
begin decr(l);
if tfm[k + l] ≠ 0 then
  begin while b > 2 do
    begin dig[j] ← a mod 8; a ← a div 8; b ← b − 3; incr(j);
    end;
  case b of
  0: a ← tfm[k + l];
  1: a ← a + 2 ∗ tfm[k + l];
  2: a ← a + 4 ∗ tfm[k + l];
  end;
  end;
b ← b + 8;
end
```

This code is used in section 58.

60. The property value may be a character, which is output in octal unless it is a letter or a digit.

```
procedure out_char(c : byte); { outputs a character }
  begin if font_type > vanilla then
    begin tfm[0] ← c; out_octal(0, 1)
    end
  else if ((c ≥ "0") ∧ (c ≤ "9")) ∨ ((c ≥ "A") ∧ (c ≤ "Z")) ∨ ((c ≥ "a") ∧ (c ≤ "z")) then
      out(´␣C␣´, xchr[c])
    else begin tfm[0] ← c; out_octal(0, 1);
      end;
  end;
```

61. The property value might be a “face” byte, which is output in the curious code mentioned earlier, provided that it is less than 18.

```
procedure out_face(k : index); { outputs a face }
  var s: 0 . . 1; { the slope }
    b: 0 . . 8; { the weight and expansion }
  begin if tfm[k] ≥ 18 then out_octal(k, 1)
  else begin out(´␣F␣´); { specify face-code format }
    s ← tfm[k] mod 2; b ← tfm[k] div 2; out(MBL_string[1 + (b mod 3)]); out(RI_string[1 + s]);
    out(RCE_string[1 + (b div 3)]);
    end;
  end;
```

62. And finally, the value might be a *fix_word*, which is output in decimal notation with just enough decimal places for `VPtoVF` to recover every bit of the given *fix_word*.

All of the numbers involved in the intermediate calculations of this procedure will be nonnegative and less than $10 \cdot 2^{24}$.

```
procedure out_fix(k : index); { outputs a fix_word }
  var a: 0 .. '7777; { accumulator for the integer part }
    f: integer; { accumulator for the fraction part }
    j: 0 .. 12; { index into dig }
    delta: integer; { amount if allowable inaccuracy }
  begin out('␣R␣'); { specify real format }
  a ← (tfm[k] * 16) + (tfm[k + 1] div 16); f ← ((tfm[k + 1] mod 16) * '400 + tfm[k + 2]) * '400 + tfm[k + 3];
  if a > '3777 then ⟨Reduce negative to positive 65⟩;
  ⟨Output the integer part, a, in decimal notation 63⟩;
  ⟨Output the fraction part, f/2^20, in decimal notation 64⟩;
  end;
```

63. The following code outputs at least one digit even if $a = 0$.

```
⟨Output the integer part, a, in decimal notation 63⟩ ≡
  begin j ← 0;
  repeat dig[j] ← a mod 10; a ← a div 10; incr(j);
  until a = 0;
  out_digs(j);
  end
```

This code is used in section 62.

64. And the following code outputs at least one digit to the right of the decimal point.

```
⟨Output the fraction part, f/2^20, in decimal notation 64⟩ ≡
  begin out('.'); f ← 10 * f + 5; delta ← 10;
  repeat if delta > '4000000 then f ← f + '2000000 − (delta div 2);
    out(f div '4000000 : 1); f ← 10 * (f mod '4000000); delta ← delta * 10;
  until f ≤ delta;
  end;
```

This code is used in section 62.

65.
```
⟨Reduce negative to positive 65⟩ ≡
  begin out('-'); a ← '10000 − a;
  if f > 0 then
    begin f ← '4000000 − f; decr(a);
    end;
  end
```

This code is used in section 62.

66. Outputting the TFM info. TEX checks the information of a `TFM` file for validity as the file is being read in, so that no further checks will be needed when typesetting is going on. And when it finds something wrong, it justs calls the file "bad," without identifying the nature of the problem, since `TFM` files are supposed to be good almost all of the time.

Of course, a bad file shows up every now and again, and that's where `VFtoVP` comes in. This program wants to catch at least as many errors as TEX does, and to give informative error messages besides. All of the errors are corrected, so that the `VPL` output will be correct (unless, of course, the `TFM` file was so loused up that no attempt is being made to fathom it).

67. Just before each character is processed, its code is printed in octal notation. Up to eight such codes appear on a line; so we have a variable to keep track of how many are currently there. We also keep track of whether or not any errors have had to be corrected.

⟨Globals in the outer block 7⟩ +≡
chars_on_line: 0 . . 8; { the number of characters printed on the current line }
perfect: *boolean*; { was the file free of errors? }

68. ⟨Set initial values 11⟩ +≡
chars_on_line ← 0;
perfect ← *true*; { innocent until proved guilty }

69. Error messages are given with the help of the *bad* and *range_error* and *bad_char* macros:

define *bad*(#) ≡
  begin *perfect* ← *false*;
  if *chars_on_line* > 0 **then** *print_ln*(´␣´);
  chars_on_line ← 0; *print_ln*(´Bad␣TFM␣file:␣´, #);
  end
define *range_error*(#) ≡
  begin *perfect* ← *false*; *print_ln*(´␣´); *print*(#, ´␣index␣for␣character␣´); *print_octal*(*c*);
  print_ln(´␣is␣too␣large;´); *print_ln*(´so␣I␣reset␣it␣to␣zero.´);
  end
define *bad_char_tail*(#) ≡ *print_octal*(#); *print_ln*(´.´);
  end
define *bad_char*(#) ≡
  begin *perfect* ← *false*;
  if *chars_on_line* > 0 **then** *print_ln*(´␣´);
  chars_on_line ← 0; *print*(´Bad␣TFM␣file:␣´, #, ´␣nonexistent␣character␣´); *bad_char_tail*
define *correct_bad_char_tail*(#) ≡ *print_octal*(*tfm*[#]); *print_ln*(´.´); *tfm*[#] ← *bc*;
  end
define *correct_bad_char*(#) ≡
  begin *perfect* ← *false*;
  if *chars_on_line* > 0 **then** *print_ln*(´␣´);
  chars_on_line ← 0; *print*(´Bad␣TFM␣file:␣´, #, ´␣nonexistent␣character␣´);
  correct_bad_char_tail

⟨Globals in the outer block 7⟩ +≡
i: 0 . . ´77777; { an index to words of a subfile }
c: 0 . . 256; { a random character }
d: 0 . . 3; { byte number in a word }
k: *index*; { a random index }
r: 0 . . 65535; { a random two-byte value }

70. There are a lot of simple things to do, and they have to be done one at a time, so we might as well get down to business. The first things that VFtoVP will put into the VPL file appear in the header part.

⟨ Do the header 70 ⟩ ≡
  begin *font_type* ← *vanilla*;
  if *lh* ≥ 12 **then**
    begin ⟨ Set the true *font_type* 75 ⟩;
    if *lh* ≥ 17 **then**
      begin ⟨ Output the family name 77 ⟩;
      if *lh* ≥ 18 **then** ⟨ Output the rest of the header 78 ⟩;
      end;
    ⟨ Output the character coding scheme 76 ⟩;
    end;
  ⟨ Output the design size 73 ⟩;
  ⟨ Output the check sum 71 ⟩;
  ⟨ Output the *seven_bit_safe_flag* 79 ⟩;
  end

This code is used in section 132.

71. ⟨ Output the check sum 71 ⟩ ≡
  left; *out*(´CHECKSUM´); *out_octal*(*check_sum*, 4); *right*

This code is used in section 70.

72. Incorrect design sizes are changed to 10 points.

  define *bad_design*(#) ≡
    begin *bad*(´Design␣size␣´, #, ´!´); *print_ln*(´I´´ve␣set␣it␣to␣10␣points.´);
    out(´␣D␣10´);
    end

73. ⟨ Output the design size 73 ⟩ ≡
  left; *out*(´DESIGNSIZE´);
  if *tfm*[*design_size*] > 127 **then** *bad_design*(´negative´)
  else if (*tfm*[*design_size*] = 0) ∧ (*tfm*[*design_size* + 1] < 16) **then** *bad_design*(´too␣small´)
    else *out_fix*(*design_size*);
  right; *out*(´(COMMENT␣DESIGNSIZE␣IS␣IN␣POINTS)´); *out_ln*;
  out(´(COMMENT␣OTHER␣SIZES␣ARE␣MULTIPLES␣OF␣DESIGNSIZE)´); *out_ln*

This code is used in section 70.

74. Since we have to check two different BCPL strings for validity, we might as well write a subroutine to make the check.

```
procedure check_BCPL(k, l : index);  { checks a string of length < l }
  var j: index;  { runs through the string }
    c: byte;  { character being checked }
  begin if tfm[k] ≥ l then
    begin bad('String␣is␣too␣long;␣I''ve␣shortened␣it␣drastically.'); tfm[k] ← 1;
    end;
  for j ← k + 1 to k + tfm[k] do
    begin c ← tfm[j];
    if (c = "(") ∨ (c = ")") then
      begin bad('Parenthesis␣in␣string␣has␣been␣changed␣to␣slash.'); tfm[j] ← "/";
      end
    else if (c < "␣") ∨ (c > "~") then
        begin bad('Nonstandard␣ASCII␣code␣has␣been␣blotted␣out.'); tfm[j] ← "?";
        end
      else if (c ≥ "a") ∧ (c ≤ "z") then tfm[j] ← c + "A" − "a";  { upper-casify letters }
    end;
  end;
```

75. The *font_type* starts out *vanilla*; possibly we need to reset it.

```
⟨ Set the true font_type 75 ⟩ ≡
  begin check_BCPL(scheme, 40);
  if (tfm[scheme] ≥ 11) ∧ (tfm[scheme + 1] = "T") ∧ (tfm[scheme + 2] = "E") ∧ (tfm[scheme + 3] = "X") ∧
        (tfm[scheme + 4] = "␣") ∧ (tfm[scheme + 5] = "M") ∧ (tfm[scheme + 6] = "A") ∧
        (tfm[scheme + 7] = "T") ∧ (tfm[scheme + 8] = "H") ∧ (tfm[scheme + 9] = "␣") then
    begin if (tfm[scheme + 10] = "S") ∧ (tfm[scheme + 11] = "Y") then font_type ← mathsy
    else if (tfm[scheme + 10] = "E") ∧ (tfm[scheme + 11] = "X") then font_type ← mathex;
    end;
  end
```

This code is used in section 70.

76. ⟨ Output the character coding scheme 76 ⟩ ≡

```
  left; out('CODINGSCHEME'); out_BCPL(scheme); right
```

This code is used in section 70.

77. ⟨ Output the family name 77 ⟩ ≡

```
  left; out('FAMILY'); check_BCPL(family, 20); out_BCPL(family); right
```

This code is used in section 70.

78. ⟨ Output the rest of the header 78 ⟩ ≡

```
  begin left; out('FACE'); out_face(random_word + 3); right;
  for i ← 18 to lh − 1 do
    begin left; out('HEADER␣D␣', i : 1); out_octal(check_sum + 4 ∗ i, 4); right;
    end;
  end
```

This code is used in section 70.

79. This program does not check to see if the *seven_bit_safe_flag* has the correct setting, i.e., if it really reflects the seven-bit-safety of the TFM file; the stated value is merely put into the VPL file. The VPtoVF program will store a correct value and give a warning message if a file falsely claims to be safe.

⟨Output the *seven_bit_safe_flag* 79⟩ ≡
if $(lh > 17) \land (tfm[random\_word] > 127)$ **then**
  begin *left*; *out*('SEVENBITSAFEFLAG␣TRUE'); *right*;
  end

This code is used in section 70.

80. The next thing to take care of is the list of parameters.

⟨Do the parameters 80⟩ ≡
if $np > 0$ **then**
  begin *left*; *out*('FONTDIMEN'); *out_ln*;
  for $i \leftarrow 1$ **to** *np* **do** ⟨Check and output the ith parameter 82⟩;
  right;
  end;
⟨Check to see if *np* is complete for this font type 81⟩;

This code is used in section 132.

81. ⟨Check to see if *np* is complete for this font type 81⟩ ≡
if $(font\_type = mathsy) \land (np \neq 22)$ **then**
  print_ln('Unusual␣number␣of␣fontdimen␣parameters␣for␣a␣math␣symbols␣font␣(', $np : 1$,
    '␣not␣22).')
else if $(font\_type = mathex) \land (np \neq 13)$ **then**
    print_ln('Unusual␣number␣of␣fontdimen␣parameters␣for␣an␣extension␣font␣(', $np : 1$,
      '␣not␣13).')

This code is used in section 80.

82. All *fix_word* values except the design size and the first parameter will be checked to make sure that they are less than 16.0 in magnitude, using the *check_fix* macro:

define *check_fix_tail*(#) ≡ *bad*(#, '␣', $i : 1$, '␣is␣too␣big;'); *print_ln*('I␣have␣set␣it␣to␣zero.');
    end
define *check_fix*(#) ≡
    if $(tfm[\#] > 0) \land (tfm[\#] < 255)$ **then**
      begin $tfm[\#] \leftarrow 0$; $tfm[(\#) + 1] \leftarrow 0$; $tfm[(\#) + 2] \leftarrow 0$; $tfm[(\#) + 3] \leftarrow 0$; *check_fix_tail*

⟨Check and output the ith parameter 82⟩ ≡
begin *left*;
if $i = 1$ **then** *out*('SLANT') { this parameter is not checked }
else begin *check_fix*(*param*(i))('Parameter');
  ⟨Output the name of parameter i 83⟩;
  end;
out_fix(*param*(i)); *right*;
end

This code is used in section 80.

83. ⟨ Output the name of parameter i 83 ⟩ ≡

```
if i ≤ 7 then
  case i of
  2: out(´SPACE´); 3: out(´STRETCH´); 4: out(´SHRINK´);
  5: out(´XHEIGHT´); 6: out(´QUAD´); 7: out(´EXTRASPACE´)
  end
else if (i ≤ 22) ∧ (font_type = mathsy) then
    case i of
    8: out(´NUM1´); 9: out(´NUM2´); 10: out(´NUM3´);
    11: out(´DENOM1´); 12: out(´DENOM2´);
    13: out(´SUP1´); 14: out(´SUP2´); 15: out(´SUP3´);
    16: out(´SUB1´); 17: out(´SUB2´);
    18: out(´SUPDROP´); 19: out(´SUBDROP´);
    20: out(´DELIM1´); 21: out(´DELIM2´);
    22: out(´AXISHEIGHT´)
    end
  else if (i ≤ 13) ∧ (font_type = mathex) then
      if i = 8 then out(´DEFAULTRULETHICKNESS´)
      else out(´BIGOPSPACING´, i − 8 : 1)
    else out(´PARAMETER␣D␣´, i : 1)
```

This code is used in section 82.

84. We need to check the range of all the remaining *fix_word* values, and to make sure that $width[0] = 0$, etc.

```
define nonzero_fix(#) ≡ (tfm[#] > 0) ∨ (tfm[# + 1] > 0) ∨ (tfm[# + 2] > 0) ∨ (tfm[# + 3] > 0)
```

⟨ Check the *fix_word* entries 84 ⟩ ≡

```
if nonzero_fix(4 ∗ width_base) then bad(´width[0]␣should␣be␣zero.´);
if nonzero_fix(4 ∗ height_base) then bad(´height[0]␣should␣be␣zero.´);
if nonzero_fix(4 ∗ depth_base) then bad(´depth[0]␣should␣be␣zero.´);
if nonzero_fix(4 ∗ italic_base) then bad(´italic[0]␣should␣be␣zero.´);
for i ← 0 to nw − 1 do check_fix(4 ∗ (width_base + i))(´Width´);
for i ← 0 to nh − 1 do check_fix(4 ∗ (height_base + i))(´Height´);
for i ← 0 to nd − 1 do check_fix(4 ∗ (depth_base + i))(´Depth´);
for i ← 0 to ni − 1 do check_fix(4 ∗ (italic_base + i))(´Italic␣correction´);
if nk > 0 then
  for i ← 0 to nk − 1 do check_fix(kern(i))(´Kern´);
```

This code is used in section 132.

85. The ligature/kerning program comes next. Before we can put it out in `VPL` format, we need to make a table of "labels" that will be inserted into the program. For each character c whose *tag* is *lig_tag* and whose starting address is r, we will store the pair (c,r) in the *label_table* array. If there's a boundary-char program starting at r, we also store the pair $(256,r)$. This array is sorted by its second components, using the simple method of straight insertion.

⟨ Globals in the outer block 7 ⟩ +≡
label_table: **array** [0 . . 258] **of record**
  cc: 0 . . 256;
  rr: 0 . . *lig_size*;
  end;
label_ptr: 0 . . 257; { the largest entry in *label_table* }
sort_ptr: 0 . . 257; { index into *label_table* }
boundary_char: 0 . . 256; { boundary character, or 256 if none }
bchar_label: 0 . . ´77777; { beginning of boundary character program }

86. ⟨ Set initial values 11 ⟩ +≡
boundary_char ← 256; *bchar_label* ← ´77777;
label_ptr ← 0; *label_table*[0].*rr* ← 0; { a sentinel appears at the bottom }

87. We'll also identify and remove inaccessible program steps, using the *activity* array.

define *unreachable* = 0 { a program step not known to be reachable }
define *pass_through* = 1 { a program step passed through on initialization }
define *accessible* = 2 { a program step that can be relevant }

⟨ Globals in the outer block 7 ⟩ +≡
activity: **array** [0 . . *lig_size*] **of** *unreachable* . . *accessible*;
ai, *acti*: 0 . . *lig_size*; { indices into *activity* }

88. ⟨ Do the ligatures and kerns 88 ⟩ ≡
if *nl* > 0 **then**
  begin for *ai* ← 0 **to** *nl* − 1 **do** *activity*[*ai*] ← *unreachable*;
  ⟨ Check for a boundary char 91 ⟩;
  end;
⟨ Build the label table 89 ⟩;
if *nl* > 0 **then**
  begin *left*; *out*(´LIGTABLE´); *out_ln*;
  ⟨ Compute the *activity* array 92 ⟩;
  ⟨ Output and correct the ligature/kern program 93 ⟩;
  right; ⟨ Check for ligature cycles 112 ⟩;
  end

This code is used in section 134.

89. We build the label table even when $nl = 0$, because this catches errors that would not otherwise be detected.

```
⟨Build the label table 89⟩ ≡
  for c ← bc to ec do
    if tag(c) = lig_tag then
      begin r ← remainder(c);
      if r < nl then
        begin if tfm[lig_step(r)] > stop_flag then
          begin r ← 256 ∗ tfm[lig_step(r) + 2] + tfm[lig_step(r) + 3];
          if r < nl then
            if activity[remainder(c)] = unreachable then activity[remainder(c)] ← pass_through;
          end;
        end;
      if r ≥ nl then
        begin perfect ← false; print_ln(´␣´);
        print(´Ligature/kern␣starting␣index␣for␣character␣´); print_octal(c);
        print_ln(´␣is␣too␣large;´); print_ln(´so␣I␣removed␣it.´); reset_tag(c);
        end
      else ⟨Insert (c, r) into label_table 90⟩;
      end;
  label_table[label_ptr + 1].rr ← lig_size;  { put "infinite" sentinel at the end }
```

This code is used in section 88.

90. ⟨Insert (c, r) into *label_table* 90⟩ ≡

```
  begin sort_ptr ← label_ptr;  { there's a hole at position sort_ptr + 1 }
  while label_table[sort_ptr].rr > r do
    begin label_table[sort_ptr + 1] ← label_table[sort_ptr]; decr(sort_ptr);  { move the hole }
    end;
  label_table[sort_ptr + 1].cc ← c; label_table[sort_ptr + 1].rr ← r;  { fill the hole }
  incr(label_ptr); activity[r] ← accessible;
  end
```

This code is used in section 89.

91. ⟨Check for a boundary char 91⟩ ≡

```
  if tfm[lig_step(0)] = 255 then
    begin left; out(´BOUNDARYCHAR´); boundary_char ← tfm[lig_step(0) + 1]; out_char(boundary_char);
    right; activity[0] ← pass_through;
    end;
  if tfm[lig_step(nl − 1)] = 255 then
    begin r ← 256 ∗ tfm[lig_step(nl − 1) + 2] + tfm[lig_step(nl − 1) + 3];
    if r ≥ nl then
      begin perfect ← false; print_ln(´␣´);
      print(´Ligature/kern␣starting␣index␣for␣boundarychar␣is␣too␣large;´);
      print_ln(´so␣I␣removed␣it.´);
      end
    else begin label_ptr ← 1; label_table[1].cc ← 256; label_table[1].rr ← r; bchar_label ← r;
      activity[r] ← accessible;
      end;
    activity[nl − 1] ← pass_through;
    end
```

This code is used in section 88.

92. ⟨ Compute the *activity* array 92 ⟩ ≡

```
for ai ← 0 to nl − 1 do
  if activity[ai] = accessible then
    begin r ← tfm[lig_step(ai)];
    if r < stop_flag then
      begin r ← r + ai + 1;
      if r ≥ nl then
        begin bad(´Ligature/kern␣step␣´, ai : 1, ´␣skips␣too␣far;´);
        print_ln(´I␣made␣it␣stop.´); tfm[lig_step(ai)] ← stop_flag;
        end
      else activity[r] ← accessible;
      end;
    end
```

This code is used in section 88.

93. We ignore *pass_through* items, which don't need to be mentioned in the VPL file.

⟨ Output and correct the ligature/kern program 93 ⟩ ≡

```
sort_ptr ← 1;  { point to the next label that will be needed }
for acti ← 0 to nl − 1 do
  if activity[acti] ≠ pass_through then
    begin i ← acti; ⟨ Take care of commenting out unreachable steps 95 ⟩;
    ⟨ Output any labels for step i 94 ⟩;
    ⟨ Output step i of the ligature/kern program 96 ⟩;
    end;
if level = 2 then right  { the final step was unreachable }
```

This code is used in section 88.

94. ⟨ Output any labels for step *i* 94 ⟩ ≡

```
while i = label_table[sort_ptr].rr do
  begin left; out(´LABEL´);
  if label_table[sort_ptr].cc = 256 then out(´␣BOUNDARYCHAR´)
  else out_char(label_table[sort_ptr].cc);
  right; incr(sort_ptr);
  end
```

This code is used in section 93.

95. ⟨ Take care of commenting out unreachable steps 95 ⟩ ≡

```
if activity[i] = unreachable then
  begin if level = 1 then
    begin left; out(´COMMENT␣THIS␣PART␣OF␣THE␣PROGRAM␣IS␣NEVER␣USED!´); out_ln;
    end
  end
else if level = 2 then right
```

This code is used in section 93.

96. ⟨ Output step i of the ligature/kern program 96 ⟩ ≡
begin $k \leftarrow lig\_step(i)$;
if $tfm[k] > stop\_flag$ **then**
begin if $256 * tfm[k+2] + tfm[k+3] \geq nl$ **then**
bad(´Ligature␣unconditional␣stop␣command␣address␣is␣too␣big.´);
end
else if $tfm[k+2] \geq kern\_flag$ **then** ⟨ Output a kern step 98 ⟩
else ⟨ Output a ligature step 99 ⟩;
if $tfm[k] > 0$ **then**
if $level = 1$ **then** ⟨ Output either `SKIP` or `STOP` 97 ⟩;
end

This code is used in sections 93 and 105.

97. The `SKIP` command is a bit tricky, because we will be omitting all inaccessible commands.

⟨ Output either `SKIP` or `STOP` 97 ⟩ ≡
begin if $tfm[k] \geq stop\_flag$ **then** *out*(´(STOP)´)
else begin $count \leftarrow 0$;
for $ai \leftarrow i+1$ **to** $i + tfm[k]$ **do**
if $activity[ai] = accessible$ **then** *incr*(*count*);
out(´(SKIP␣D␣´, *count* : 1, ´)´); { possibly $count = 0$, so who cares }
end;
out_ln;
end

This code is used in section 96.

98. ⟨ Output a kern step 98 ⟩ ≡
begin if $nonexistent(tfm[k+1])$ **then**
if $tfm[k+1] \neq boundary\_char$ **then** *correct_bad_char*(´Kern␣step␣for´)$(k+1)$;
left; *out*(´KRN´); *out_char*$(tfm[k+1])$; $r \leftarrow 256 * (tfm[k+2] - kern\_flag) + tfm[k+3]$;
if $r \geq nk$ **then**
begin *bad*(´Kern␣index␣too␣large.´); *out*(´␣R␣0.0´);
end
else *out_fix*$(kern(r))$;
right;
end

This code is used in section 96.

99. ⟨ Output a ligature step 99 ⟩ ≡

```
begin if nonexistent(tfm[k + 1]) then
  if tfm[k + 1] ≠ boundary_char then correct_bad_char(´Ligature␣step␣for´)(k + 1);
if nonexistent(tfm[k + 3]) then correct_bad_char(´Ligature␣step␣produces␣the´)(k + 3);
left; r ← tfm[k + 2];
if (r = 4) ∨ ((r > 7) ∧ (r ≠ 11)) then
  begin print_ln(´Ligature␣step␣with␣nonstandard␣code␣changed␣to␣LIG´); r ← 0; tfm[k+2] ← 0;
  end;
if r mod 4 > 1 then out(´/´);
out(´LIG´);
if odd(r) then out(´/´);
while r > 3 do
  begin out(´>´); r ← r − 4;
  end;
out_char(tfm[k + 1]); out_char(tfm[k + 3]); right;
end
```

This code is used in section 96.

100. The last thing on VFtoVP's agenda is to go through the list of *char_info* and spew out the information about each individual character.

⟨ Do the characters 100 ⟩ ≡

```
sort_ptr ← 0;  { this will suppress ‘STOP’ lines in ligature comments }
for c ← bc to ec do
  if width_index(c) > 0 then
    begin if chars_on_line = 8 then
      begin print_ln(´␣´); chars_on_line ← 1;
      end
    else begin if chars_on_line > 0 then print(´␣´);
      incr(chars_on_line);
      end;
    print_octal(c);  { progress report }
    left; out(´CHARACTER´); out_char(c); out_ln; ⟨ Output the character's width 101 ⟩;
    if height_index(c) > 0 then ⟨ Output the character's height 102 ⟩;
    if depth_index(c) > 0 then ⟨ Output the character's depth 103 ⟩;
    if italic_index(c) > 0 then ⟨ Output the italic correction 104 ⟩;
    case tag(c) of
    no_tag: do_nothing;
    lig_tag: ⟨ Output the applicable part of the ligature/kern program as a comment 105 ⟩;
    list_tag: ⟨ Output the character link unless there is a problem 106 ⟩;
    ext_tag: ⟨ Output an extensible character recipe 107 ⟩;
    end;
    if ¬do_map(c) then goto final_end;
    right;
    end
```

This code is used in section 133.

101. ⟨ Output the character's width 101 ⟩ ≡
  begin *left*; *out*(`'CHARWD'`);
  if $width\_index(c) \ge nw$ **then** *range_error*(`'Width'`)
  else *out_fix*(*width*(*c*));
  right;
  end

This code is used in section 100.

102. ⟨ Output the character's height 102 ⟩ ≡
  if $height\_index(c) \ge nh$ **then** *range_error*(`'Height'`)
  else begin *left*; *out*(`'CHARHT'`); *out_fix*(*height*(*c*)); *right*;
    end

This code is used in section 100.

103. ⟨ Output the character's depth 103 ⟩ ≡
  if $depth\_index(c) \ge nd$ **then** *range_error*(`'Depth'`)
  else begin *left*; *out*(`'CHARDP'`); *out_fix*(*depth*(*c*)); *right*;
    end

This code is used in section 100.

104. ⟨ Output the italic correction 104 ⟩ ≡
  if $italic\_index(c) \ge ni$ **then** *range_error*(`'Italic␣correction'`)
  else begin *left*; *out*(`'CHARIC'`); *out_fix*(*italic*(*c*)); *right*;
    end

This code is used in section 100.

105. ⟨ Output the applicable part of the ligature/kern program as a comment 105 ⟩ ≡
  begin *left*; *out*(`'COMMENT'`); *out_ln*;
  $i \leftarrow remainder(c)$; $r \leftarrow lig\_step(i)$;
  if $tfm[r] > stop\_flag$ **then** $i \leftarrow 256 * tfm[r+2] + tfm[r+3]$;
  repeat ⟨ Output step *i* of the ligature/kern program 96 ⟩;
    if $tfm[k] \ge stop\_flag$ **then** $i \leftarrow nl$
    else $i \leftarrow i + 1 + tfm[k]$;
  until $i \ge nl$;
  right;
  end

This code is used in section 100.

106. We want to make sure that there is no cycle of characters linked together by *list_tag* entries, since such a cycle would get TeX into an endless loop. If such a cycle exists, the routine here detects it when processing the largest character code in the cycle.

⟨Output the character link unless there is a problem 106⟩ ≡
```
begin r ← remainder(c);
if nonexistent(r) then
  begin bad_char(´Character␣list␣link␣to´)(r); reset_tag(c);
  end
else begin while (r < c) ∧ (tag(r) = list_tag) do r ← remainder(r);
  if r = c then
    begin bad(´Cycle␣in␣a␣character␣list!´); print(´Character␣´); print_octal(c);
    print_ln(´␣now␣ends␣the␣list.´); reset_tag(c);
    end
  else begin left; out(´NEXTLARGER´); out_char(remainder(c)); right;
    end;
  end;
end
```
This code is used in section 100.

107. ⟨Output an extensible character recipe 107⟩ ≡
```
if remainder(c) ≥ ne then
  begin range_error(´Extensible´); reset_tag(c);
  end
else begin left; out(´VARCHAR´); out_ln; ⟨Output the extensible pieces that exist 108⟩;
  right;
  end
```
This code is used in section 100.

108. ⟨Output the extensible pieces that exist 108⟩ ≡
```
for k ← 0 to 3 do
  if (k = 3) ∨ (tfm[exten(c) + k] > 0) then
    begin left;
    case k of
    0: out(´TOP´); 1: out(´MID´); 2: out(´BOT´); 3: out(´REP´)
    end;
    if nonexistent(tfm[exten(c) + k]) then out_char(c)
    else out_char(tfm[exten(c) + k]);
    right;
    end
```
This code is used in section 107.

109. Some of the extensible recipes may not actually be used, but TeX will complain about them anyway if they refer to nonexistent characters. Therefore `VFtoVP` must check them too.

⟨ Check the extensible recipes 109 ⟩ ≡
if $ne > 0$ **then**
  for $c \leftarrow 0$ **to** $ne - 1$ **do**
    for $d \leftarrow 0$ **to** 3 **do**
      begin $k \leftarrow 4 * (exten\_base + c) + d$;
      if $(tfm[k] > 0) \lor (d = 3)$ **then**
        begin if $nonexistent(tfm[k])$ **then**
          begin *bad_char*(`´Extensible␣recipe␣involves␣the´`)($tfm[k]$);
          if $d < 3$ **then** $tfm[k] \leftarrow 0$;
          end;
        end;
      end

This code is used in section 134.

110. Checking for ligature loops. We have programmed almost everything but the most interesting calculation of all, which has been saved for last as a special treat. TEX's extended ligature mechanism allows unwary users to specify sequences of ligature replacements that never terminate. For example, the pair of commands

(/LIG x y) (/LIG y x)

alternately replaces character x by character y and vice versa. A similar loop occurs if (LIG/ z y) occurs in the program for x and (LIG/ z x) occurs in the program for y.

More complicated loops are also possible. For example, suppose the ligature programs for x and y are

(LABEL x)(/LIG/ z w)(/LIG/> w y) $\ldots,$
(LABEL y)(LIG w x) $\ldots;$

then the adjacent characters xz change to xwz, $xywz$, xxz, $xxwz$, $\ldots$, ad infinitum.

111. To detect such loops, VFtoVP attempts to evaluate the function $f(x,y)$ for all character pairs x and y, where f is defined as follows: If the current character is x and the next character is y, we say the "cursor" is between x and y; when the cursor first moves past y, the character immediately to its left is $f(x,y)$. This function is defined if and only if no infinite loop is generated when the cursor is between x and y.

The function $f(x,y)$ can be defined recursively. It turns out that all pairs (x,y) belong to one of five classes. The simplest class has $f(x,y)=y$; this happens if there's no ligature between x and y, or in the cases LIG/> and /LIG/>>. Another simple class arises when there's a LIG or /LIG> between x and y, generating the character z; then $f(x,y)=z$. Otherwise we always have $f(x,y)$ equal to either $f(x,z)$ or $f(z,y)$ or $f(f(x,z),y)$, where z is the inserted ligature character.

The first two of these classes can be merged; we can also consider (x,y) to belong to the simple class when $f(x,y)$ has been evaluated. For technical reasons we allow x to be 256 (for the boundary character at the left) or 257 (in cases when an error has been detected).

For each pair (x,y) having a ligature program step, we store (x,y) in a hash table from which the values z and *class* can be read.

define *simple* $= 0$ $\{ f(x,y)=z \}$
define *left_z* $= 1$ $\{ f(x,y)=f(z,y) \}$
define *right_z* $= 2$ $\{ f(x,y)=f(x,z) \}$
define *both_z* $= 3$ $\{ f(x,y)=f(f(x,z),y) \}$
define *pending* $= 4$ $\{ f(x,y)$ is being evaluated $\}$

⟨Globals in the outer block 7⟩ +≡
hash: **array** [0 .. *hash_size*] **of** 0 .. 66048; $\{ 256x+y+1$ for $x \le 257$ and $y \le 255 \}$
class: **array** [0 .. *hash_size*] **of** *simple* .. *pending*;
lig_z: **array** [0 .. *hash_size*] **of** 0 .. 257;
hash_ptr: 0 .. *hash_size*; { the number of nonzero entries in *hash* }
hash_list: **array** [0 .. *hash_size*] **of** 0 .. *hash_size*; { list of those nonzero entries }
h, *hh*: 0 .. *hash_size*; { indices into the hash table }
x_lig_cycle, *y_lig_cycle*: 0 .. 256; { problematic ligature pair }

112. ⟨ Check for ligature cycles 112 ⟩ ≡

```
hash_ptr ← 0; y_lig_cycle ← 256;
for hh ← 0 to hash_size do hash[hh] ← 0; { clear the hash table }
for c ← bc to ec do
  if tag(c) = lig_tag then
    begin i ← remainder(c);
    if tfm[lig_step(i)] > stop_flag then i ← 256 * tfm[lig_step(i) + 2] + tfm[lig_step(i) + 3];
    ⟨ Enter data for character c starting at location i in the hash table 113 ⟩;
    end;
if bchar_label < nl then
  begin c ← 256; i ← bchar_label;
  ⟨ Enter data for character c starting at location i in the hash table 113 ⟩;
  end;
if hash_ptr = hash_size then
  begin print_ln('Sorry,␣I␣haven''t␣room␣for␣so␣many␣ligature/kern␣pairs!'); goto final_end;
  end;
for hh ← 1 to hash_ptr do
  begin r ← hash_list[hh];
  if class[r] > simple then { make sure f is defined }
    r ← f(r, (hash[r] − 1) div 256, (hash[r] − 1) mod 256);
  end;
if y_lig_cycle < 256 then
  begin print('Infinite␣ligature␣loop␣starting␣with␣');
  if x_lig_cycle = 256 then print('boundary') else print_octal(x_lig_cycle);
  print('␣and␣'); print_octal(y_lig_cycle); print_ln('!');
  out('(INFINITE␣LIGATURE␣LOOP␣MUST␣BE␣BROKEN!)'); goto final_end;
  end
```

This code is used in section 88.

113. ⟨ Enter data for character c starting at location i in the hash table 113 ⟩ ≡

```
repeat hash_input; k ← tfm[lig_step(i)];
  if k ≥ stop_flag then i ← nl
  else i ← i + 1 + k;
until i ≥ nl
```

This code is used in sections 112 and 112.

114. We use an "ordered hash table" with linear probing, because such a table is efficient when the lookup of a random key tends to be unsuccessful.

```
procedure hash_input;  { enter data for character c and command i }
  label exit;
  var cc: simple .. both_z;  { class of data being entered }
    zz: 0 .. 255;  { function value or ligature character being entered }
    y: 0 .. 255;  { the character after the cursor }
    key: integer;  { value to be stored in hash }
    t: integer;  { temporary register for swapping }
  begin if hash_ptr = hash_size then return;
  ⟨ Compute the command parameters y, cc, and zz 115 ⟩;
  key ← 256 * c + y + 1;  h ← (1009 * key) mod hash_size;
  while hash[h] > 0 do
    begin if hash[h] ≤ key then
      begin if hash[h] = key then return;  { unused ligature command }
      t ← hash[h];  hash[h] ← key;  key ← t;  { do ordered-hash-table insertion }
      t ← class[h];  class[h] ← cc;  cc ← t;  { namely, do a swap }
      t ← lig_z[h];  lig_z[h] ← zz;  zz ← t;
      end;
    if h > 0 then decr(h) else h ← hash_size;
    end;
  hash[h] ← key;  class[h] ← cc;  lig_z[h] ← zz;  incr(hash_ptr);  hash_list[hash_ptr] ← h;
exit: end;
```

115. We must store kern commands as well as ligature commands, because the former might make the latter inapplicable.

```
⟨ Compute the command parameters y, cc, and zz 115 ⟩ ≡
  k ← lig_step(i);  y ← tfm[k + 1];  t ← tfm[k + 2];  cc ← simple;  zz ← tfm[k + 3];
  if t ≥ kern_flag then zz ← y
  else begin case t of
    0, 6: do_nothing;  { LIG,/LIG> }
    5, 11: zz ← y;  { LIG/>, /LIG/>> }
    1, 7: cc ← left_z;  { LIG/, /LIG/> }
    2: cc ← right_z;  { /LIG }
    3: cc ← both_z;  { /LIG/ }
    end;  { there are no other cases }
    end
```

This code is used in section 114.

116. Evaluation of $f(x,y)$ is handled by two mutually recursive procedures. Kind of a neat algorithm, generalizing a depth-first search.

```
function f(h, x, y : index): index;  forward;  { compute f for arguments known to be in hash[h] }
function eval(x, y : index): index;  { compute f(x, y) with hashtable lookup }
  var key: integer;  { value sought in hash table }
  begin key ← 256 * x + y + 1;  h ← (1009 * key) mod hash_size;
  while hash[h] > key do
    if h > 0 then decr(h) else h ← hash_size;
  if hash[h] < key then eval ← y  { not in ordered hash table }
  else eval ← f(h, x, y);
  end;
```

117. Pascal's beastly convention for *forward* declarations prevents us from saying **function** $f(h, x, y : index)$: *index* here.

function f;
  begin case $class[h]$ **of**
  $simple$: $do\_nothing$;
  $left\_z$: **begin** $class[h] \leftarrow pending$; $lig\_z[h] \leftarrow eval(lig\_z[h], y)$; $class[h] \leftarrow simple$;
    end;
  $right\_z$: **begin** $class[h] \leftarrow pending$; $lig\_z[h] \leftarrow eval(x, lig\_z[h])$; $class[h] \leftarrow simple$;
    end;
  $both\_z$: **begin** $class[h] \leftarrow pending$; $lig\_z[h] \leftarrow eval(eval(x, lig\_z[h]), y)$; $class[h] \leftarrow simple$;
    end;
  $pending$: **begin** $x\_lig\_cycle \leftarrow x$; $y\_lig\_cycle \leftarrow y$; $lig\_z[h] \leftarrow 257$; $class[h] \leftarrow simple$;
    end; { the value 257 will break all cycles, since it's not in *hash* }
  end; { there are no other cases }
  $f \leftarrow lig\_z[h]$;
  end;

118. Outputting the VF info. The routines we've used for output from the *tfm* array have counterparts for output from *vf*. One difference is that the string outputs from *vf* need to be checked for balanced parentheses. The *string_balance* routine tests the string of length l that starts at location k.

```
function string_balance(k, l : integer): boolean;
  label not_found, exit;
  var j, bal: integer;
  begin if l > 0 then
    if vf[k] = "␣" then goto not_found;   { a leading blank is considered unbalanced }
  bal ← 0;
  for j ← k to k + l − 1 do
    begin if (vf[j] < "␣") ∨ (vf[j] ≥ 127) then goto not_found;
    if vf[j] = "(" then incr(bal)
    else if vf[j] = ")" then
        if bal = 0 then goto not_found
        else decr(bal);
    end;
  if bal > 0 then goto not_found;
  string_balance ← true; return;
not_found: string_balance ← false;
exit: end;
```

119. **define** *bad_vf*(#) ≡

```
        begin perfect ← false;
        if chars_on_line > 0 then print_ln(´␣´);
        chars_on_line ← 0; print_ln(´Bad␣VF␣file:␣´, #);
        end
```

⟨ Do the virtual font title 119 ⟩ ≡

```
  if string_balance(0, font_start[0]) then
    begin left; out(´VTITLE␣´);
    for k ← 0 to font_start[0] − 1 do out(xchr[vf[k]]);
    right;
    end
  else bad_vf(´Title␣is␣not␣a␣balanced␣ASCII␣string´)
```

This code is used in section 132.

120. We can re-use some code by moving *fix_word* data to *tfm*, using the fact that the design size has already been output.

```
procedure out_as_fix(x : integer);
  var k: 1 .. 3;
  begin if abs(x) ≥ ´100000000 then
    begin bad_vf(´Oversize␣dimension␣has␣been␣reset␣to␣zero.´); x ← 0;
    end;
  if x ≥ 0 then tfm[design_size] ← 0
  else begin tfm[design_size] ← 255; x ← x + ´100000000;
    end;
  for k ← 3 downto 1 do
    begin tfm[design_size + k] ← x mod 256; x ← x div 256;
    end;
  out_fix(design_size);
  end;
```

121. ⟨ Do the local fonts 121 ⟩ ≡
for $f \leftarrow 0$ **to** $font\_ptr - 1$ **do**
  begin *left*; *out*(`´MAPFONT␣D␣´`, $f:1$); *out_ln*; ⟨ Output the font area and name 122 ⟩;
  for $k \leftarrow 0$ **to** 11 **do** $tfm[k] \leftarrow vf[font\_start[f]+k]$;
  if $tfm[0]+tfm[1]+tfm[2]+tfm[3]>0$ **then**
    begin *left*; *out*(`´FONTCHECKSUM´`); *out_octal*(0, 4); *right*;
    end;
  left; *out*(`´FONTAT´`); *out_fix*(4); *right*; *left*; *out*(`´FONTDSIZE´`); *out_fix*(8); *right*; *right*;
  end

This code is used in section 132.

122. ⟨ Output the font area and name 122 ⟩ ≡
$a \leftarrow vf[font\_start[f]+12]$; $l \leftarrow vf[font\_start[f]+13]$;
if $a>0$ **then**
  if $\neg string\_balance(font\_start[f]+14, a)$ **then** *bad_vf*(`´Improper␣font␣area␣will␣be␣ignored´`)
  else begin *left*; *out*(`´FONTAREA␣´`);
    for $k \leftarrow font\_start[f]+14$ **to** $font\_start[f]+a+13$ **do** $out(xchr[vf[k]])$;
    right;
    end;
if $(l=0) \lor \neg string\_balance(font\_start[f]+14+a, l)$ **then**
  bad_vf(`´Improper␣font␣name␣will␣be␣ignored´`)
else begin *left*; *out*(`´FONTNAME␣´`);
  for $k \leftarrow font\_start[f]+14+a$ **to** $font\_start[f]+a+l+13$ **do** $out(xchr[vf[k]])$;
  right;
  end

This code is used in section 121.

123. Now we get to the interesting part of `VF` output, where `DVI` commands are translated into symbolic form. The `VPL` language is a subset of `DVI`, so we sometimes need to output semantic equivalents of the commands instead of producing a literal translation. This causes a small but tolerable loss of efficiency. We need to simulate the stack used by `DVI`-reading software.

⟨ Globals in the outer block 7 ⟩ +≡
top: 0 .. *max_stack*; { `DVI` stack pointer }
wstack, *xstack*, *ystack*, *zstack*: **array** [0 .. *max_stack*] **of** *integer*;
  { stacked values of `DVI` registers w, x, y, z }
vf_limit: 0 .. *vf_size*; { the current packet ends here }
o: *byte*; { the current opcode }

124. ⟨ Do the packet for character c 124 ⟩ ≡

```
if packet_start[c] = vf_size then bad_vf('Missing␣packet␣for␣character␣', c : 1)
else begin left; out('MAP'); out_ln; top ← 0; wstack[0] ← 0; xstack[0] ← 0; ystack[0] ← 0;
  zstack[0] ← 0; vf_ptr ← packet_start[c]; vf_limit ← packet_end[c] + 1; f ← 0;
  while vf_ptr < vf_limit do
    begin o ← vf[vf_ptr]; incr(vf_ptr);
    case o of
      ⟨ Cases of DVI instructions that can appear in character packets 126 ⟩
    improper_DVI_for_VF: bad_vf('Illegal␣DVI␣code␣', o : 1, '␣will␣be␣ignored');
    end; { there are no other cases }
    end;
  if top > 0 then
    begin bad_vf('More␣pushes␣than␣pops!');
    repeat out('(POP)'); decr(top); until top = 0;
    end;
  right;
  end
```

This code is used in section 133.

125. A procedure called *get_bytes* helps fetch the parameters of DVI commands.

```
function get_bytes(k : integer; signed : boolean): integer;
  var a: integer; { accumulator }
  begin if vf_ptr + k > vf_limit then
    begin bad_vf('Packet␣ended␣prematurely'); k ← vf_limit − vf_ptr;
    end;
  a ← vf[vf_ptr];
  if (k = 4) ∨ signed then
    if a ≥ 128 then a ← a − 256;
  incr(vf_ptr);
  while k > 1 do
    begin a ← a ∗ 256 + vf[vf_ptr]; incr(vf_ptr); decr(k);
    end;
  get_bytes ← a;
  end;
```

126. Let's look at the simplest cases first, in order to get some experience.

define *four_cases*(#) ≡ #, # + 1, # + 2, # + 3
define *eight_cases*(#) ≡ *four_cases*(#), *four_cases*(# + 4)
define *sixteen_cases*(#) ≡ *eight_cases*(#), *eight_cases*(# + 8)
define *thirty_two_cases*(#) ≡ *sixteen_cases*(#), *sixteen_cases*(# + 16)
define *sixty_four_cases*(#) ≡ *thirty_two_cases*(#), *thirty_two_cases*(# + 32)

⟨ Cases of DVI instructions that can appear in character packets 126 ⟩ ≡

```
nop: do_nothing;
push: begin if top = max_stack then
    begin print_ln('Stack␣overflow!'); goto final_end;
    end;
  incr(top); wstack[top] ← wstack[top − 1]; xstack[top] ← xstack[top − 1]; ystack[top] ← ystack[top − 1];
  zstack[top] ← zstack[top − 1]; out('(PUSH)'); out_ln;
  end;
pop: if top = 0 then bad_vf('More␣pops␣than␣pushes!')
  else begin decr(top); out('(POP)'); out_ln;
    end;
set_rule, put_rule: begin if o = put_rule then out('(PUSH)');
  left; out('SETRULE'); out_as_fix(get_bytes(4, true)); out_as_fix(get_bytes(4, true));
  if o = put_rule then out(')(POP');
  right;
  end;
```

See also sections 127, 128, 129, and 130.

This code is used in section 124.

127. Horizontal and vertical motions become RIGHT and DOWN in VPL lingo.

⟨ Cases of DVI instructions that can appear in character packets 126 ⟩ +≡

```
four_cases(right1): begin out('(MOVERIGHT'); out_as_fix(get_bytes(o − right1 + 1, true)); out(')');
  out_ln; end;
w0, four_cases(w1): begin if o ≠ w0 then wstack[top] ← get_bytes(o − w1 + 1, true);
  out('(MOVERIGHT'); out_as_fix(wstack[top]); out(')'); out_ln; end;
x0, four_cases(x1): begin if o ≠ x0 then xstack[top] ← get_bytes(o − x1 + 1, true);
  out('(MOVERIGHT'); out_as_fix(xstack[top]); out(')'); out_ln; end;
four_cases(down1): begin out('(MOVEDOWN'); out_as_fix(get_bytes(o − down1 + 1, true)); out(')');
  out_ln; end;
y0, four_cases(y1): begin if o ≠ y0 then ystack[top] ← get_bytes(o − y1 + 1, true);
  out('(MOVEDOWN'); out_as_fix(ystack[top]); out(')'); out_ln; end;
z0, four_cases(z1): begin if o ≠ z0 then zstack[top] ← get_bytes(o − z1 + 1, true);
  out('(MOVEDOWN'); out_as_fix(zstack[top]); out(')'); out_ln; end;
```

128. Variable f always refers to the current font. If $f = font\_ptr$, it's a font that hasn't been defined (so its characters will be ignored).

⟨ Cases of DVI instructions that can appear in character packets 126 ⟩ +≡

```
sixty_four_cases(fnt_num_0), four_cases(fnt1): begin f ← 0;
  if o ≥ fnt1 then font_number[font_ptr] ← get_bytes(o − fnt1 + 1, false)
  else font_number[font_ptr] ← o − fnt_num_0;
  while font_number[f] ≠ font_number[font_ptr] do incr(f);
  if f = font_ptr then bad_vf('Undeclared␣font␣selected')
  else begin out('(SELECTFONT␣D␣', f : 1, ')'); out_ln;
    end;
  end;
```

129. Before we typeset a character we make sure that it exists.

⟨ Cases of DVI instructions that can appear in character packets 126 ⟩ +≡

```
sixty_four_cases(set_char_0), sixty_four_cases(set_char_0 + 64), four_cases(set1), four_cases(put1): begin if
      o ≥ set1 then
    if o ≥ put1 then c ← get_bytes(o − put1 + 1, false)
    else c ← get_bytes(o − set1 + 1, false)
  else c ← o;
  if f = font_ptr then bad_vf('Character␣', c : 1, '␣in␣undeclared␣font␣will␣be␣ignored')
  else begin vf[font_start[f + 1] − 1] ← c;   { store c in the "hole" we left }
    k ← font_chars[f]; while vf[k] ≠ c do incr(k);
    if k = font_start[f + 1] − 1 then bad_vf('Character␣', c : 1, '␣in␣font␣', f : 1, '␣will␣be␣ignored')
    else begin if o ≥ put1 then out('(PUSH)');
      left; out('SETCHAR'); out_char(c);
      if o ≥ put1 then out(')(POP');
      right;
      end;
    end;
  end;
```

130. The "special" commands are the only ones remaining to be dealt with. We use a hexadecimal output in the general case, if a simple string would be inadequate.

```
  define out_hex(#) ≡
          begin a ← #;
          if a < 10 then out(a : 1)
          else out(xchr[a − 10 + "A"]);
          end
```

⟨ Cases of DVI instructions that can appear in character packets 126 ⟩ +≡

```
four_cases(xxx1): begin k ← get_bytes(o − xxx1 + 1, false);
  if k < 0 then bad_vf('String␣of␣negative␣length!')
  else begin left;
    if k + vf_ptr > vf_limit then
      begin bad_vf('Special␣command␣truncated␣to␣packet␣length'); k ← vf_limit − vf_ptr;
      end;
    if (k > 64) ∨ ¬string_balance(vf_ptr, k) then
      begin out('SPECIALHEX␣');
      while k > 0 do
        begin if k mod 32 = 0 then out_ln
        else if k mod 4 = 0 then out('␣');
        out_hex(vf[vf_ptr] div 16); out_hex(vf[vf_ptr] mod 16); incr(vf_ptr); decr(k);
        end;
      end
    else begin out('SPECIAL␣');
      while k > 0 do
        begin out(xchr[vf[vf_ptr]]); incr(vf_ptr); decr(k);
        end;
      end;
    right;
    end;
  end;
```

131. The main program. The routines sketched out so far need to be packaged into separate procedures, on some systems, since some Pascal compilers place a strict limit on the size of a routine. The packaging is done here in an attempt to avoid some system-dependent changes.

First come the *vf_input* and *organize* procedures, which read the input data and get ready for subsequent events. If something goes wrong, the routines return *false*.

```
function vf_input: boolean;
  label final_end, exit;
  var vf_ptr: 0 .. vf_size; { an index into vf }
    k: integer; { all-purpose index }
    c: integer; { character code }
  begin ⟨Read the whole VF file 31⟩;
  vf_input ← true; return;
final_end: vf_input ← false;
exit: end;

function organize: boolean;
  label final_end, exit;
  var tfm_ptr: index; { an index into tfm }
  begin ⟨Read the whole TFM file 24⟩;
  ⟨Set subfile sizes lh, bc, ..., np 25⟩;
  ⟨Compute the base addresses 27⟩;
  organize ← vf_input; return;
final_end: organize ← false;
exit: end;
```

132. Next we do the simple things.

```
procedure do_simple_things;
  var i: 0 .. '77777; { an index to words of a subfile }
    f: 0 .. vf_size; { local font number }
    k: integer; { all-purpose index }
  begin ⟨Do the virtual font title 119⟩;
  ⟨Do the header 70⟩;
  ⟨Do the parameters 80⟩;
  ⟨Do the local fonts 121⟩;
  ⟨Check the fix_word entries 84⟩;
  end;
```

133. And then there's a routine for individual characters.

function *do_map*(*c* : *byte*): *boolean*;
  label *final_end*, *exit*;
  var *k*: *integer*; *f*: 0 . . *vf_size*; { current font number }
  begin ⟨ Do the packet for character *c* 124 ⟩;
  do_map ← *true*; **return**;
final_end: *do_map* ← *false*;
exit: **end**;

function *do_characters*: *boolean*;
  label *final_end*, *exit*;
  var *c*: *byte*; { character being done }
    k: *index*; { a random index }
    ai: 0 . . *lig_size*; { index into *activity* }
  begin ⟨ Do the characters 100 ⟩;
  do_characters ← *true*; **return**;
final_end: *do_characters* ← *false*;
exit: **end**;

134. Here is where `VFtoVP` begins and ends.

  begin *initialize*;
  if ¬*organize* **then goto** *final_end*;
  do_simple_things;
  ⟨ Do the ligatures and kerns 88 ⟩;
  ⟨ Check the extensible recipes 109 ⟩;
  if ¬*do_characters* **then goto** *final_end*;
  print_ln(´.´);
  if *level* ≠ 0 **then** *print_ln*(´This␣program␣isn´´t␣working!´);
  if ¬*perfect* **then**
    begin *out*(´(COMMENT␣THE␣TFM␣AND/OR␣VF␣FILE␣WAS␣BAD,␣´);
    out(´SO␣THE␣DATA␣HAS␣BEEN␣CHANGED!)´); *write_ln*(*vpl_file*);
    end;
final_end: **end**.

135. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make `VFtoVP` work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

136. Index. Pointers to error messages appear here together with the section numbers where each identifier is used.

⟨ Read and store a font definition 35 ⟩ Used in section 33.
⟨ Read and store the font definitions and character packets 33 ⟩ Used in section 31.
⟨ Read and verify the postamble 34 ⟩ Used in section 31.
⟨ Read the local font's `TFM` file and record the characters it contains 39 ⟩ Used in section 35.
⟨ Read the preamble command 32 ⟩ Used in section 31.
⟨ Read the whole `TFM` file 24 ⟩ Used in section 131.
⟨ Read the whole `VF` file 31 ⟩ Used in section 131.
⟨ Reduce l by one, preserving the invariants 59 ⟩ Used in section 58.
⟨ Reduce negative to positive 65 ⟩ Used in section 62.
⟨ Set initial values 11, 21, 43, 50, 55, 68, 86 ⟩ Used in section 2.
⟨ Set subfile sizes lh, bc, $\ldots$, np 25 ⟩ Used in section 131.
⟨ Set the true *font_type* 75 ⟩ Used in section 70.
⟨ Take care of commenting out unreachable steps 95 ⟩ Used in section 93.
⟨ Types in the outer block 5, 22 ⟩ Used in section 2.

The VFtoVP processor

(Version 1.3, December 2002)

The preparation of this program was supported in part by the National Science Foundation and by the System Development Foundation. 'TEX' is a trademark of the American Mathematical Society.