The VPtoVF processor

(Version 1.5, August 1998)

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.

1. Introduction. The `VPtoVF` utility program converts virtual-property-list ("`VPL`") files into an equivalent pair of files called a virtual font ("`VF`") file and a TEX font metric ("`TFM`") file. It also makes a thorough check of the given `VPL` file, so that the `VF` file should be acceptable to device drivers and the `TFM` file should be acceptable to TEX.

`VPtoVF` is an extended version of the program `PLtoTF`, 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 `PLFONT`.

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

define *banner* ≡ ´`This␣is␣VPtoVF,␣Version␣1.5`´ { printed when the program starts }

2. This program is written entirely in standard Pascal, except that it has to do some slightly system-dependent character code conversion on input. Furthermore, lower case letters are used in error messages; they could be converted to upper case if necessary. The input is read from *vpl_file*, and the output is written on *vf_file* and *tfm_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 *VPtoVF*(*vpl_file*, *vf_file*, *tfm_file*, *output*);
const ⟨ Constants in the outer block 3 ⟩
type ⟨ Types in the outer block 23 ⟩
var ⟨ Globals in the outer block 5 ⟩
procedure *initialize*; { this procedure gets things started properly }
var ⟨ Local variables for initialization 25 ⟩
begin *print_ln*(*banner*);
⟨ Set initial values 6 ⟩
end;

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

⟨ Constants in the outer block 3 ⟩ ≡
buf_size = 60; { length of lines displayed in error messages }
max_header_bytes = 100; { four times the maximum number of words allowed in the `TFM` file header block, must be 1024 or less }
vf_size = 10000; { maximum length of *vf* data, in bytes }
max_stack = 100; { maximum depth of simulated `DVI` stack }
max_param_words = 30; { the maximum number of `fontdimen` parameters allowed }
max_lig_steps = 5000; { maximum length of ligature program, must be at most 32767 − 257 = 32510 }
max_kerns = 500; { the maximum number of distinct kern values }
hash_size = 5003;
{ preferably a prime number, a bit larger than the number of character pairs in lig/kern steps }

This code is used in section 2.

4. Here are some macros for common programming idioms.

define *incr*(#) ≡ # ← # + 1 { increase a variable by unity }
define *decr*(#) ≡ # ← # − 1 { decrease a variable by unity }
define *do_nothing* ≡ { empty statement }

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

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

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

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

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

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

This code is used in section 2.

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

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

This code is used in section 2.

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

A PL file is a list of entries of the form

```
(PROPERTYNAME VALUE)
```

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

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

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

The example ligature table is a bit trickier. It specifies that the letter f followed by another f is changed to code *´200*, while code *´200* followed by i is changed to *´201*; presumably codes *´200* and *´201* represent the ligatures 'ff' and 'ffi'. Moreover, in both cases f and *´200*, if the following character is the code *´51* (which is

a right parenthesis), an additional 1.5 units of space should be inserted before the *´51*. (The '`SKIP D 1`' skips over one `LIG` or `KRN` command, which in this case is the second `LIG`; in this way two different ligature/kern programs can come together.) Finally, if either `f` or *´200* is followed by a question mark, the question mark is replaced by `f` and the ligature program is started over. (Thus, the character pair '`f?`' would actually become the ligature 'ff', and '`ff?`' or '`f?f`' would become 'fff'. To avoid this restart procedure, the `/LIG` command could be replaced by `/LIG>`; then '`f?`' would become 'ff' and '`f?f`' would become 'fff'.)

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

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

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

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

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

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

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

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

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

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

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

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

Finally, to typeset character *´201*, we do something that requires a special ability to interpret PostScript commands; this example sets the PostScript "color" to 50% gray and typesets an 'A' from cmr10 at 20\u in that color.

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

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

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

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

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

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

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

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

SEVENBITSAFEFLAG (string value, default is 'FALSE'). The value should start with either 'T' (true) or 'F' (false). If true, character codes less than 128 cannot lead to codes of 128 or more via ligatures or charlists or extensible characters. (TEX82 ignores this flag, but older versions of TEX would only accept TFM files that were seven-bit safe.) VPtoVF computes the correct value of this flag and gives an error message only if a claimed "true" value is incorrect.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`LIG` `/LIG` `/LIG>` `LIG/` `LIG/>` `/LIG/` `/LIG/>` `/LIG/>>`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

20. So that is what `VPL` files hold. From these rules, you can guess (correctly) that `VPtoVF` operates in four main stages. First it assigns the default values to all properties; then it scans through the `VPL` file, changing property values as new ones are seen; then it checks the information and corrects any problems; and finally it outputs the `VF` and `TFM` files.

21. The next question is, "What are `VF` and `TFM` files?" A complete answer to that question appears in the documentation of the companion programs, `VFtoVP` and `TFtoPL`, so the details will not be repeated here. Suffice it to say that a `VF` or `TFM` file stores all of the relevant font information in a sequence of 8-bit bytes. The number of bytes is always a multiple of 4, so we could regard the files as sequences of 32-bit words; but TEX uses the byte interpretation, and so does `VPtoVF`. Note that the bytes are considered to be unsigned numbers.

```
⟨ Globals in the outer block 5 ⟩ +≡
vf_file: packed file of 0 .. 255;
tfm_file: packed file of 0 .. 255;
```

22. On some systems you may have to do something special to write 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 6 ⟩ +≡
  rewrite(vf_file); rewrite(tfm_file);
```

23. Basic input routines. For the purposes of this program, a *byte* is an unsigned eight-bit quantity, and an *ASCII_code* is an integer between ´40 and ´177. Such ASCII codes correspond to one-character constants like "A" in WEB language.

⟨ Types in the outer block 23 ⟩ ≡
```
byte = 0 .. 255;  { unsigned eight-bit quantity }
ASCII_code = ´40 .. ´177;  { standard ASCII code numbers }
```
See also sections 66, 71, 78, and 81.

This code is used in section 2.

24. One of the things VPtoVF has to do is convert characters of strings to ASCII form, since that is the code used for the family name and the coding scheme in a TFM file. An array *xord* is used to do the conversion from *char*; the method below should work with little or no change on most Pascal systems.

```
define first_ord = 0  { ordinal number of the smallest element of char }
define last_ord = 127  { ordinal number of the largest element of char }
```
⟨ Globals in the outer block 5 ⟩ +≡
```
xord: array [char] of ASCII_code;  { conversion table }
```

25. ⟨ Local variables for initialization 25 ⟩ ≡
```
k: integer;  { all-purpose initialization index }
```
See also sections 48, 79, and 83.

This code is used in section 2.

26. Characters that should not appear in VPL files (except in comments) are mapped into ´177.

```
define invalid_code = ´177  { code deserving an error message }
```
⟨ Set initial values 6 ⟩ +≡
```
for k ← first_ord to last_ord do xord[chr(k)] ← invalid_code;
xord[´␣´] ← "␣"; xord[´!´] ← "!"; xord[´"´] ← """"; xord[´#´] ← "#"; xord[´$´] ← "$";
xord[´%´] ← "%"; xord[´&´] ← "&"; xord[´´´] ← "´"; xord[´(´] ← "("; xord[´)´] ← ")";
xord[´*´] ← "*"; xord[´+´] ← "+"; xord[´,´] ← ","; xord[´-´] ← "-"; xord[´.´] ← ".";
xord[´/´] ← "/"; xord[´0´] ← "0"; xord[´1´] ← "1"; xord[´2´] ← "2"; xord[´3´] ← "3";
xord[´4´] ← "4"; xord[´5´] ← "5"; xord[´6´] ← "6"; xord[´7´] ← "7"; xord[´8´] ← "8";
xord[´9´] ← "9"; xord[´:´] ← ":"; xord[´;´] ← ";"; xord[´<´] ← "<"; xord[´=´] ← "=";
xord[´>´] ← ">"; xord[´?´] ← "?"; xord[´@´] ← "@"; xord[´A´] ← "A"; xord[´B´] ← "B";
xord[´C´] ← "C"; xord[´D´] ← "D"; xord[´E´] ← "E"; xord[´F´] ← "F"; xord[´G´] ← "G";
xord[´H´] ← "H"; xord[´I´] ← "I"; xord[´J´] ← "J"; xord[´K´] ← "K"; xord[´L´] ← "L";
xord[´M´] ← "M"; xord[´N´] ← "N"; xord[´O´] ← "O"; xord[´P´] ← "P"; xord[´Q´] ← "Q";
xord[´R´] ← "R"; xord[´S´] ← "S"; xord[´T´] ← "T"; xord[´U´] ← "U"; xord[´V´] ← "V";
xord[´W´] ← "W"; xord[´X´] ← "X"; xord[´Y´] ← "Y"; xord[´Z´] ← "Z"; xord[´[´] ← "[";
xord[´\´] ← "\"; xord[´]´] ← "]"; xord[´^´] ← "^"; xord[´_´] ← "_"; xord[´`´] ← "`";
xord[´a´] ← "a"; xord[´b´] ← "b"; xord[´c´] ← "c"; xord[´d´] ← "d"; xord[´e´] ← "e";
xord[´f´] ← "f"; xord[´g´] ← "g"; xord[´h´] ← "h"; xord[´i´] ← "i"; xord[´j´] ← "j";
xord[´k´] ← "k"; xord[´l´] ← "l"; xord[´m´] ← "m"; xord[´n´] ← "n"; xord[´o´] ← "o";
xord[´p´] ← "p"; xord[´q´] ← "q"; xord[´r´] ← "r"; xord[´s´] ← "s"; xord[´t´] ← "t";
xord[´u´] ← "u"; xord[´v´] ← "v"; xord[´w´] ← "w"; xord[´x´] ← "x"; xord[´y´] ← "y";
xord[´z´] ← "z"; xord[´{´] ← "{"; xord[´|´] ← "|"; xord[´}´] ← "}"; xord[´~´] ← "~";
```

27. In order to help catch errors of badly nested parentheses, `VPtoVF` assumes that the user will begin each line with a number of blank spaces equal to some constant times the number of open parentheses at the beginning of that line. However, the program doesn't know in advance what the constant is, nor does it want to print an error message on every line for a user who has followed no consistent pattern of indentation.

Therefore the following strategy is adopted: If the user has been consistent with indentation for ten or more lines, an indentation error will be reported. The constant of indentation is reset on every line that should have nonzero indentation.

⟨ Globals in the outer block 5 ⟩ +≡
line: *integer*; { the number of the current line }
good_indent: *integer*; { the number of lines since the last bad indentation }
indent: *integer*; { the number of spaces per open parenthesis, zero if unknown }
level: *integer*; { the current number of open parentheses }

28. ⟨ Set initial values 6 ⟩ +≡
line ← 0; *good_indent* ← 0; *indent* ← 0; *level* ← 0;

29. The input need not really be broken into lines of any maximum length, and we could read it character by character without any buffering. But we shall place it into a small buffer so that offending lines can be displayed in error messages.

⟨ Globals in the outer block 5 ⟩ +≡
left_ln, *right_ln*: *boolean*; { are the left and right ends of the buffer at end-of-line marks? }
limit: 0 .. *buf_size*; { position of the last character present in the buffer }
loc: 0 .. *buf_size*; { position of the last character read in the buffer }
buffer: **array** [1 .. *buf_size*] **of** *char*;
input_has_ended: *boolean*; { there is no more input to read }

30. ⟨ Set initial values 6 ⟩ +≡
limit ← 0; *loc* ← 0; *left_ln* ← *true*; *right_ln* ← *true*; *input_has_ended* ← *false*;

31. Just before each `CHARACTER` property list is evaluated, the character 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.

⟨ Globals in the outer block 5 ⟩ +≡
chars_on_line: 0 .. 8; { the number of characters printed on the current line }

32. ⟨ Set initial values 6 ⟩ +≡
chars_on_line ← 0;

33. The following routine prints an error message and an indication of where the error was detected. The error message should not include any final punctuation, since this procedure supplies its own.

```
define err_print(#) ≡
        begin if chars_on_line > 0 then print_ln(´␣´);
        print(#); show_error_context;
        end
procedure show_error_context; { prints the current scanner location }
  var k: 0 .. buf_size; { an index into buffer }
  begin print_ln(´␣(line␣´, line : 1, ´).´);
  if ¬left_ln then print(´...´);
  for k ← 1 to loc do print(buffer[k]); { print the characters already scanned }
  print_ln(´␣´);
  if ¬left_ln then print(´␣␣␣´);
  for k ← 1 to loc do print(´␣´); { space out the second line }
  for k ← loc + 1 to limit do print(buffer[k]); { print the characters yet unseen }
  if right_ln then print_ln(´␣´) else print_ln(´...´);
  chars_on_line ← 0;
  end;
```

34. Here is a procedure that does the right thing when we are done reading the present contents of the buffer. It keeps *buffer*[*buf_size*] empty, in order to avoid range errors on certain Pascal compilers.

An infinite sequence of right parentheses is placed at the end of the file, so that the program is sure to get out of whatever level of nesting it is in.

On some systems it is desirable to modify this code so that tab marks in the buffer are replaced by blank spaces. (Simply setting *xord*[*chr*(´11´)] ← "␣" would not work; for example, two-line error messages would not come out properly aligned.)

```
procedure fill_buffer;
  begin left_ln ← right_ln; limit ← 0; loc ← 0;
  if left_ln then
    begin if line > 0 then read_ln(vpl_file);
    incr(line);
    end;
  if eof(vpl_file) then
    begin limit ← 1; buffer[1] ← ´)´; right_ln ← false; input_has_ended ← true;
    end
  else begin while (limit < buf_size − 1) ∧ (¬eoln(vpl_file)) do
      begin incr(limit); read(vpl_file, buffer[limit]);
      end;
    buffer[limit + 1] ← ´␣´; right_ln ← eoln(vpl_file);
    if left_ln then ⟨Set loc to the number of leading blanks in the buffer, and check the indentation 35⟩;
    end;
  end;
```

35. The interesting part about *fill_buffer* is the part that learns what indentation conventions the user is following, if any.

```
define bad_indent(#) ≡
        begin if good_indent ≥ 10 then err_print(#);
        good_indent ← 0; indent ← 0;
        end
```

⟨ Set *loc* to the number of leading blanks in the buffer, and check the indentation 35 ⟩ ≡

```
begin while (loc < limit) ∧ (buffer[loc + 1] = ´␣´) do incr(loc);
if loc < limit then
  begin if level = 0 then
    if loc = 0 then incr(good_indent)
    else bad_indent(´Warning:␣Indented␣line␣occurred␣at␣level␣zero´)
  else if indent = 0 then
      if loc mod level = 0 then
        begin indent ← loc div level; good_indent ← 1;
        end
      else good_indent ← 0
    else if indent ∗ level = loc then incr(good_indent)
      else bad_indent(´Warning:␣Inconsistent␣indentation;␣´,
            ´you␣are␣at␣parenthesis␣level␣´, level : 1);
  end;
end
```

This code is used in section 34.

36. Basic scanning routines. The global variable *cur_char* holds the ASCII code corresponding to the character most recently read from the input buffer, or to a character that has been substituted for the real one.

```
⟨ Globals in the outer block 5 ⟩ +≡
cur_char: ASCII_code;  { we have just read this }
```

37. Here is a procedure that sets *cur_char* to an ASCII code for the next character of input, if that character is a letter or digit or slash or `>`. Otherwise it sets *cur_char* ← `"␣"`, and the input system will be poised to reread the character that was rejected, whether or not it was a space. Lower case letters are converted to upper case.

```
procedure get_keyword_char;
  begin while (loc = limit) ∧ (¬right_ln) do fill_buffer;
  if loc = limit then cur_char ← "␣"   { end-of-line counts as a delimiter }
  else begin cur_char ← xord[buffer[loc + 1]];
    if cur_char ≥ "a" then cur_char ← cur_char − ´40;
    if ((cur_char ≥ "0") ∧ (cur_char ≤ "9")) then incr(loc)
    else if ((cur_char ≥ "A") ∧ (cur_char ≤ "Z")) then incr(loc)
      else if cur_char = "/" then incr(loc)
        else if cur_char = ">" then incr(loc)
          else cur_char ← "␣";
    end;
  end;
```

38. The following procedure sets *cur_char* to the next character code, and converts lower case to upper case. If the character is a left or right parenthesis, it will not be "digested"; the character will be read again and again, until the calling routine does something like '*incr*(*loc*)' to get past it. Such special treatment of parentheses insures that the structural information they contain won't be lost in the midst of other error recovery operations.

```
  define backup ≡
          begin if (cur_char > ")") ∨ (cur_char < "(") then decr(loc);
          end   { undoes the effect of get_next }
procedure get_next;   { sets cur_char to next, balks at parentheses }
  begin while loc = limit do fill_buffer;
  incr(loc); cur_char ← xord[buffer[loc]];
  if cur_char ≥ "a" then
    if cur_char ≤ "z" then cur_char ← cur_char − ´40   { uppercasify }
    else begin if cur_char = invalid_code then
        begin err_print(´Illegal␣character␣in␣the␣file´); cur_char ← "?";
        end;
      end
  else if (cur_char ≤ ")") ∧ (cur_char ≥ "(") then decr(loc);
  end;
```

39. Here's a procedure that scans a hexadecimal digit or a right parenthesis.

```
function get_hex: byte;
  var a: integer;  { partial result }
  begin repeat get_next;
  until cur_char ≠ "␣";
  a ← cur_char − ")";
  if a > 0 then
    begin a ← cur_char − "0";
    if cur_char > "9" then
      if cur_char < "A" then a ← −1
      else a ← cur_char − "A" + 10;
    end;
  if (a < 0) ∨ (a > 15) then
    begin err_print('Illegal␣hexadecimal␣digit'); get_hex ← 0;
    end
  else get_hex ← a;
  end;
```

40. The next procedure is used to ignore the text of a comment, or to pass over erroneous material. As such, it has the privilege of passing parentheses. It stops after the first right parenthesis that drops the level below the level in force when the procedure was called.

```
procedure skip_to_end_of_item;
  var l: integer;  { initial value of level }
  begin l ← level;
  while level ≥ l do
    begin while loc = limit do fill_buffer;
    incr(loc);
    if buffer[loc] = ')' then decr(level)
    else if buffer[loc] = '(' then incr(level);
    end;
  if input_has_ended then err_print('File␣ended␣unexpectedly:␣No␣closing␣")"');
  cur_char ← "␣";  { now the right parenthesis has been read and digested }
  end;
```

41. A similar procedure copies the bytes remaining in an item. The copied bytes go into an array *vf* that we'll declare later. Leading blanks are ignored.

```
  define vf_store(#) ≡
          begin vf[vf_ptr] ← #;
          if vf_ptr = vf_size then err_print('I''m␣out␣of␣memory---increase␣my␣vfsize!')
          else incr(vf_ptr);
          end
procedure copy_to_end_of_item;
  label 30;
  var l: integer;  { initial value of level }
    nonblank_found: boolean;  { have we seen a nonblank character yet? }
  begin l ← level; nonblank_found ← false;
  while true do
    begin while loc = limit do fill_buffer;
    if buffer[loc + 1] = ')' then
      if level = l then goto 30 else decr(level);
    incr(loc);
    if buffer[loc] = '(' then incr(level);
    if buffer[loc] ≠ '␣' then nonblank_found ← true;
    if nonblank_found then
      if xord[buffer[loc]] = invalid_code then
        begin err_print('Illegal␣character␣in␣the␣file'); vf_store("?");
        end
      else vf_store(xord[buffer[loc]]);
    end;
30: end;
```

42. Sometimes we merely want to skip past characters in the input until we reach a left or a right parenthesis. For example, we do this whenever we have finished scanning a property value and we hope that a right parenthesis is next (except for possible blank spaces).

```
  define skip_to_paren ≡
          repeat get_next until (cur_char = "(") ∨ (cur_char = ")")
  define skip_error(#) ≡
          begin err_print(#); skip_to_paren;
          end  { this gets to the right parenthesis if something goes wrong }
  define flush_error(#) ≡
          begin err_print(#); skip_to_end_of_item;
          end  { this gets past the right parenthesis if something goes wrong }
```

43. After a property value has been scanned, we want to move just past the right parenthesis that should come next in the input (except for possible blank spaces).

```
procedure finish_the_property;  { do this when the value has been scanned }
  begin while cur_char = "␣" do get_next;
  if cur_char ≠ ")" then err_print('Junk␣after␣property␣value␣will␣be␣ignored');
  skip_to_end_of_item;
  end;
```

44. Scanning property names. We have to figure out the meaning of names that appear in the `VPL` file, by looking them up in a dictionary of known keywords. Keyword number n appears in locations $start[n]$ through $start[n+1]-1$ of an array called *dictionary*.

define *max_name_index* = 100 { upper bound on the number of keywords }
define *max_letters* = 666 { upper bound on the total length of all keywords }

⟨ Globals in the outer block 5 ⟩ +≡
start: **array** [1 .. *max_name_index*] **of** 0 .. *max_letters*;
dictionary: **array** [0 .. *max_letters*] **of** *ASCII_code*;
start_ptr: 0 .. *max_name_index*; { the first available place in *start* }
dict_ptr: 0 .. *max_letters*; { the first available place in *dictionary* }

45. ⟨ Set initial values 6 ⟩ +≡
start_ptr ← 1; *start*[1] ← 0; *dict_ptr* ← 0;

46. When we are looking for a name, we put it into the *cur_name* array. When we have found it, the corresponding *start* index will go into the global variable *name_ptr*.

define *longest_name* = 20 { length of `DEFAULTRULETHICKNESS` }

⟨ Globals in the outer block 5 ⟩ +≡
cur_name: **array** [1 .. *longest_name*] **of** *ASCII_code*; { a name to look up }
name_length: 0 .. *longest_name*; { its length }
name_ptr: 0 .. *max_name_index*; { its ordinal number in the dictionary }

47. A conventional hash table with linear probing (cf. Algorithm 6.4L in *The Art of Computer Programming*) is used for the dictionary operations. If $nhash[h] = 0$, the table position is empty, otherwise $nhash[h]$ points into the *start* array.

define *hash_prime* = 141 { size of the hash table }

⟨ Globals in the outer block 5 ⟩ +≡
nhash: **array** [0 .. *hash_prime* − 1] **of** 0 .. *max_name_index*;
cur_hash: 0 .. *hash_prime* − 1; { current position in the hash table }

48. ⟨ Local variables for initialization 25 ⟩ +≡
h: 0 .. *hash_prime* − 1; { runs through the hash table }

49. ⟨ Set initial values 6 ⟩ +≡
for *h* ← 0 **to** *hash_prime* − 1 **do** *nhash*[*h*] ← 0;

50. Since there is no chance of the hash table overflowing, the procedure is very simple. After *lookup* has done its work, *cur_hash* will point to the place where the given name was found, or where it should be inserted.

```
procedure lookup; { finds cur_name in the dictionary }
  var k: 0 .. longest_name; { index into cur_name }
    j: 0 .. max_letters; { index into dictionary }
    not_found: boolean; { clumsy thing necessary to avoid goto statement }
  begin ⟨ Compute the hash code, cur_hash, for cur_name 51 ⟩;
  not_found ← true;
  while not_found do
    begin if cur_hash = 0 then cur_hash ← hash_prime − 1 else decr(cur_hash);
    if nhash[cur_hash] = 0 then not_found ← false
    else begin j ← start[nhash[cur_hash]];
      if start[nhash[cur_hash] + 1] = j + name_length then
        begin not_found ← false;
        for k ← 1 to name_length do
          if dictionary[j + k − 1] ≠ cur_name[k] then not_found ← true;
        end;
      end;
    end;
  name_ptr ← nhash[cur_hash];
  end;
```

51. ⟨ Compute the hash code, *cur_hash*, for *cur_name* 51 ⟩ ≡

```
  cur_hash ← cur_name[1];
  for k ← 2 to name_length do cur_hash ← (cur_hash + cur_hash + cur_name[k]) mod hash_prime
```

This code is used in section 50.

52. The "meaning" of the keyword that begins at *start*[*k*] in the dictionary is kept in *equiv*[*k*]. The numeric *equiv* codes are given symbolic meanings by the following definitions.

define *comment_code* = 0
define *check_sum_code* = 1
define *design_size_code* = 2
define *design_units_code* = 3
define *coding_scheme_code* = 4
define *family_code* = 5
define *face_code* = 6
define *seven_bit_safe_flag_code* = 7
define *header_code* = 8
define *font_dimen_code* = 9
define *lig_table_code* = 10
define *boundary_char_code* = 11
define *virtual_title_code* = 12
define *map_font_code* = 13
define *character_code* = 14
define *font_name_code* = 20
define *font_area_code* = 21
define *font_checksum_code* = 22
define *font_at_code* = 23
define *font_dsize_code* = 24
define *parameter_code* = 30
define *char_info_code* = 60
define *width* = 1
define *height* = 2
define *depth* = 3
define *italic* = 4
define *char_wd_code* = *char_info_code* + *width*
define *char_ht_code* = *char_info_code* + *height*
define *char_dp_code* = *char_info_code* + *depth*
define *char_ic_code* = *char_info_code* + *italic*
define *next_larger_code* = 65
define *map_code* = 66
define *var_char_code* = 67
define *select_font_code* = 80
define *set_char_code* = 81
define *set_rule_code* = 82
define *move_right_code* = 83
define *move_down_code* = 85
define *push_code* = 87
define *pop_code* = 88
define *special_code* = 89
define *special_hex_code* = 90
define *label_code* = 100
define *stop_code* = 101
define *skip_code* = 102
define *krn_code* = 103
define *lig_code* = 104

⟨ Globals in the outer block 5 ⟩ +≡
equiv: **array** [0 .. *max_name_index*] **of** *byte*;
cur_code: *byte*; { equivalent most recently found in *equiv* }

53. We have to get the keywords into the hash table and into the dictionary in the first place (sigh). The procedure that does this has the desired *equiv* code as a parameter. In order to facilitate `WEB` macro writing for the initialization, the keyword being initialized is placed into the last positions of *cur_name*, instead of the first positions.

```
procedure enter_name(v : byte);  { cur_name goes into the dictionary }
  var k: 0 .. longest_name;
  begin for k ← 1 to name_length do cur_name[k] ← cur_name[k + longest_name − name_length];
        { now the name has been shifted into the correct position }
  lookup;  { this sets cur_hash to the proper insertion place }
  nhash[cur_hash] ← start_ptr; equiv[start_ptr] ← v;
  for k ← 1 to name_length do
    begin dictionary[dict_ptr] ← cur_name[k]; incr(dict_ptr);
    end;
  incr(start_ptr); start[start_ptr] ← dict_ptr;
  end;
```

54. Here are the macros to load a name of up to 20 letters into the dictionary. For example, the macro *load5* is used for five-letter keywords.

```
define tail(#) ≡ enter_name(#)
define t20(#) ≡ cur_name[20] ← #; tail
define t19(#) ≡ cur_name[19] ← #; t20
define t18(#) ≡ cur_name[18] ← #; t19
define t17(#) ≡ cur_name[17] ← #; t18
define t16(#) ≡ cur_name[16] ← #; t17
define t15(#) ≡ cur_name[15] ← #; t16
define t14(#) ≡ cur_name[14] ← #; t15
define t13(#) ≡ cur_name[13] ← #; t14
define t12(#) ≡ cur_name[12] ← #; t13
define t11(#) ≡ cur_name[11] ← #; t12
define t10(#) ≡ cur_name[10] ← #; t11
define t9(#) ≡ cur_name[9] ← #; t10
define t8(#) ≡ cur_name[8] ← #; t9
define t7(#) ≡ cur_name[7] ← #; t8
define t6(#) ≡ cur_name[6] ← #; t7
define t5(#) ≡ cur_name[5] ← #; t6
define t4(#) ≡ cur_name[4] ← #; t5
define t3(#) ≡ cur_name[3] ← #; t4
define t2(#) ≡ cur_name[2] ← #; t3
define t1(#) ≡ cur_name[1] ← #; t2
define load3 ≡ name_length ← 3; t18
define load4 ≡ name_length ← 4; t17
define load5 ≡ name_length ← 5; t16
define load6 ≡ name_length ← 6; t15
define load7 ≡ name_length ← 7; t14
define load8 ≡ name_length ← 8; t13
define load9 ≡ name_length ← 9; t12
define load10 ≡ name_length ← 10; t11
define load11 ≡ name_length ← 11; t10
define load12 ≡ name_length ← 12; t9
define load13 ≡ name_length ← 13; t8
define load14 ≡ name_length ← 14; t7
define load15 ≡ name_length ← 15; t6
define load16 ≡ name_length ← 16; t5
define load17 ≡ name_length ← 17; t4
define load18 ≡ name_length ← 18; t3
define load19 ≡ name_length ← 19; t2
define load20 ≡ name_length ← 20; t1
```

55. (Thank goodness for keyboard macros in the text editor used to create this WEB file.)

⟨ Enter all the PL names and their equivalents, except the parameter names 55 ⟩ ≡

```
equiv[0] ← comment_code;  { this is used after unknown keywords }
load8("C")("H")("E")("C")("K")("S")("U")("M")(check_sum_code);
load10("D")("E")("S")("I")("G")("N")("S")("I")("Z")("E")(design_size_code);
load11("D")("E")("S")("I")("G")("N")("U")("N")("I")("T")("S")(design_units_code);
load12("C")("O")("D")("I")("N")("G")("S")("C")("H")("E")("M")("E")(coding_scheme_code);
load6("F")("A")("M")("I")("L")("Y")(family_code);
load4("F")("A")("C")("E")(face_code);
load16("S")("E")("V")("E")("N")("B")("I")("T")
    ("S")("A")("F")("E")("F")("L")("A")("G")(seven_bit_safe_flag_code);
load6("H")("E")("A")("D")("E")("R")(header_code);
load9("F")("O")("N")("T")("D")("I")("M")("E")("N")(font_dimen_code);
load8("L")("I")("G")("T")("A")("B")("L")("E")(lig_table_code);
load12("B")("O")("U")("N")("D")("A")("R")("Y")("C")("H")("A")("R")(boundary_char_code);
load9("C")("H")("A")("R")("A")("C")("T")("E")("R")(character_code);
load9("P")("A")("R")("A")("M")("E")("T")("E")("R")(parameter_code);
load6("C")("H")("A")("R")("W")("D")(char_wd_code);
load6("C")("H")("A")("R")("H")("T")(char_ht_code);
load6("C")("H")("A")("R")("D")("P")(char_dp_code);
load6("C")("H")("A")("R")("I")("C")(char_ic_code);
load10("N")("E")("X")("T")("L")("A")("R")("G")("E")("R")(next_larger_code);
load7("V")("A")("R")("C")("H")("A")("R")(var_char_code);
load3("T")("O")("P")(var_char_code + 1);
load3("M")("I")("D")(var_char_code + 2);
load3("B")("O")("T")(var_char_code + 3);
load3("R")("E")("P")(var_char_code + 4);
load3("E")("X")("T")(var_char_code + 4);  { compatibility with older PL format }
load7("C")("O")("M")("M")("E")("N")("T")(comment_code);
load5("L")("A")("B")("E")("L")(label_code);
load4("S")("T")("O")("P")(stop_code);
load4("S")("K")("I")("P")(skip_code);
load3("K")("R")("N")(krn_code);
load3("L")("I")("G")(lig_code);
load4("/")("L")("I")("G")(lig_code + 2);
load5("/")("L")("I")("G")(">")(lig_code + 6);
load4("L")("I")("G")("/")(lig_code + 1);
load5("L")("I")("G")("/")(">")(lig_code + 5);
load5("/")("L")("I")("G")("/")(lig_code + 3);
load6("/")("L")("I")("G")("/")(">")(lig_code + 7);
load7("/")("L")("I")("G")("/")(">")(">")(lig_code + 11);
```

This code is used in section 180.

56. VPL files may contain the following in addition to the PL names.

⟨ Enter all the VPL names 56 ⟩ ≡

```
load6("V")("T")("I")("T")("L")("E")(virtual_title_code);
load7("M")("A")("P")("F")("O")("N")("T")(map_font_code);
load3("M")("A")("P")(map_code);
load8("F")("O")("N")("T")("N")("A")("M")("E")(font_name_code);
load8("F")("O")("N")("T")("A")("R")("E")("A")(font_area_code);
load12("F")("O")("N")("T")("C")("H")("E")("C")("K")("S")("U")("M")(font_checksum_code);
load6("F")("O")("N")("T")("A")("T")(font_at_code);
load9("F")("O")("N")("T")("D")("S")("I")("Z")("E")(font_dsize_code);
load10("S")("E")("L")("E")("C")("T")("F")("O")("N")("T")(select_font_code);
load7("S")("E")("T")("C")("H")("A")("R")(set_char_code);
load7("S")("E")("T")("R")("U")("L")("E")(set_rule_code);
load9("M")("O")("V")("E")("R")("I")("G")("H")("T")(move_right_code);
load8("M")("O")("V")("E")("L")("E")("F")("T")(move_right_code + 1);
load8("M")("O")("V")("E")("D")("O")("W")("N")(move_down_code);
load6("M")("O")("V")("E")("U")("P")(move_down_code + 1);
load4("P")("U")("S")("H")(push_code);
load3("P")("O")("P")(pop_code);
load7("S")("P")("E")("C")("I")("A")("L")(special_code);
load10("S")("P")("E")("C")("I")("A")("L")("H")("E")("X")(special_hex_code);
```

This code is used in section 180.

57. ⟨ Enter the parameter names 57 ⟩ ≡

```
load5("S")("L")("A")("N")("T")(parameter_code + 1);
load5("S")("P")("A")("C")("E")(parameter_code + 2);
load7("S")("T")("R")("E")("T")("C")("H")(parameter_code + 3);
load6("S")("H")("R")("I")("N")("K")(parameter_code + 4);
load7("X")("H")("E")("I")("G")("H")("T")(parameter_code + 5);
load4("Q")("U")("A")("D")(parameter_code + 6);
load10("E")("X")("T")("R")("A")("S")("P")("A")("C")("E")(parameter_code + 7);
load4("N")("U")("M")("1")(parameter_code + 8);
load4("N")("U")("M")("2")(parameter_code + 9);
load4("N")("U")("M")("3")(parameter_code + 10);
load6("D")("E")("N")("O")("M")("1")(parameter_code + 11);
load6("D")("E")("N")("O")("M")("2")(parameter_code + 12);
load4("S")("U")("P")("1")(parameter_code + 13);
load4("S")("U")("P")("2")(parameter_code + 14);
load4("S")("U")("P")("3")(parameter_code + 15);
load4("S")("U")("B")("1")(parameter_code + 16);
load4("S")("U")("B")("2")(parameter_code + 17);
load7("S")("U")("P")("D")("R")("O")("P")(parameter_code + 18);
load7("S")("U")("B")("D")("R")("O")("P")(parameter_code + 19);
load6("D")("E")("L")("I")("M")("1")(parameter_code + 20);
load6("D")("E")("L")("I")("M")("2")(parameter_code + 21);
load10("A")("X")("I")("S")("H")("E")("I")("G")("H")("T")(parameter_code + 22);
load20("D")("E")("F")("A")("U")("L")("T")("R")("U")("L")("E")
    ("T")("H")("I")("C")("K")("N")("E")("S")("S")(parameter_code + 8);
load13("B")("I")("G")("O")("P")("S")("P")("A")("C")("I")("N")("G")("1")(parameter_code + 9);
load13("B")("I")("G")("O")("P")("S")("P")("A")("C")("I")("N")("G")("2")(parameter_code + 10);
load13("B")("I")("G")("O")("P")("S")("P")("A")("C")("I")("N")("G")("3")(parameter_code + 11);
load13("B")("I")("G")("O")("P")("S")("P")("A")("C")("I")("N")("G")("4")(parameter_code + 12);
load13("B")("I")("G")("O")("P")("S")("P")("A")("C")("I")("N")("G")("5")(parameter_code + 13);
```

This code is used in section 180.

58. When a left parenthesis has been scanned, the following routine is used to interpret the keyword that follows, and to store the equivalent value in *cur_code*.

```
procedure get_name;
  begin incr(loc); incr(level);   { pass the left parenthesis }
  cur_char ← "␣";
  while cur_char = "␣" do get_next;
  if (cur_char > ")") ∨ (cur_char < "(") then decr(loc);   { back up one character }
  name_length ← 0; get_keyword_char;   { prepare to scan the name }
  while cur_char ≠ "␣" do
    begin if name_length = longest_name then cur_name[1] ← "X"   { force error }
    else incr(name_length);
    cur_name[name_length] ← cur_char; get_keyword_char;
    end;
  lookup;
  if name_ptr = 0 then err_print('Sorry,␣I␣don''t␣know␣that␣property␣name');
  cur_code ← equiv[name_ptr];
  end;
```

59. Scanning numeric data. The next thing we need is a trio of subroutines to read the one-byte, four-byte, and real numbers that may appear as property values. These subroutines are careful to stick to numbers between -2^{31} and $2^{31}-1$, inclusive, so that a computer with two's complement 32-bit arithmetic will not be interrupted by overflow.

60. The first number scanner, which returns a one-byte value, surely has no problems of arithmetic overflow.

```
function get_byte: byte;  { scans a one-byte property value }
  var acc: integer;  { an accumulator }
    t: ASCII_code;  { the type of value to be scanned }
  begin repeat get_next;
  until cur_char ≠ "␣";  { skip the blanks before the type code }
  t ← cur_char; acc ← 0;
  repeat get_next;
  until cur_char ≠ "␣";  { skip the blanks after the type code }
  if t = "C" then ⟨ Scan an ASCII character code 61 ⟩
  else if t = "D" then ⟨ Scan a small decimal number 62 ⟩
    else if t = "O" then ⟨ Scan a small octal number 63 ⟩
      else if t = "H" then ⟨ Scan a small hexadecimal number 64 ⟩
        else if t = "F" then ⟨ Scan a face code 65 ⟩
          else skip_error('You␣need␣"C"␣or␣"D"␣or␣"O"␣or␣"H"␣or␣"F"␣here');
  cur_char ← "␣"; get_byte ← acc;
  end;
```

61. The *get_next* routine converts lower case to upper case, but it leaves the character in the buffer, so we can unconvert it.

```
⟨ Scan an ASCII character code 61 ⟩ ≡
  if (cur_char ≥ '41) ∧ (cur_char ≤ '176) ∧ ((cur_char < "(") ∨ (cur_char > ")")) then
    acc ← xord[buffer[loc]]
  else skip_error('"C"␣value␣must␣be␣standard␣ASCII␣and␣not␣a␣paren')
```

This code is used in section 60.

62.

```
⟨ Scan a small decimal number 62 ⟩ ≡
  begin while (cur_char ≥ "0") ∧ (cur_char ≤ "9") do
    begin acc ← acc * 10 + cur_char − "0";
    if acc > 255 then
      begin skip_error('This␣value␣shouldn''t␣exceed␣255'); acc ← 0; cur_char ← "␣";
      end
    else get_next;
    end;
  backup;
  end
```

This code is used in section 60.

63. ⟨ Scan a small octal number 63 ⟩ ≡

```
begin while (cur_char ≥ "0") ∧ (cur_char ≤ "7") do
  begin acc ← acc * 8 + cur_char − "0";
  if acc > 255 then
    begin skip_error(´This␣value␣shouldn´´t␣exceed␣´´377´); acc ← 0; cur_char ← "␣";
    end
  else get_next;
  end;
backup;
end
```

This code is used in section 60.

64. ⟨ Scan a small hexadecimal number 64 ⟩ ≡

```
begin while ((cur_char ≥ "0") ∧ (cur_char ≤ "9")) ∨ ((cur_char ≥ "A") ∧ (cur_char ≤ "F")) do
  begin if cur_char ≥ "A" then cur_char ← cur_char + "0" + 10 − "A";
  acc ← acc * 16 + cur_char − "0";
  if acc > 255 then
    begin skip_error(´This␣value␣shouldn´´t␣exceed␣"FF´); acc ← 0; cur_char ← "␣";
    end
  else get_next;
  end;
backup;
end
```

This code is used in section 60.

65. ⟨ Scan a face code 65 ⟩ ≡

```
begin if cur_char = "B" then acc ← 2
else if cur_char = "L" then acc ← 4
  else if cur_char ≠ "M" then acc ← 18;
get_next;
if cur_char = "I" then incr(acc)
else if cur_char ≠ "R" then acc ← 18;
get_next;
if cur_char = "C" then acc ← acc + 6
else if cur_char = "E" then acc ← acc + 12
  else if cur_char ≠ "R" then acc ← 18;
if acc ≥ 18 then
  begin skip_error(´Illegal␣face␣code,␣I␣changed␣it␣to␣MRR´); acc ← 0;
  end;
end
```

This code is used in section 60.

66. The routine that scans a four-byte value puts its output into *cur_bytes*, which is a record containing (yes, you guessed it) four bytes.

⟨ Types in the outer block 23 ⟩ +≡

```
four_bytes = record b0: byte; b1: byte; b2: byte; b3: byte;
  end;
```

67. **define** *c0* ≡ *cur_bytes.b0*
define *c1* ≡ *cur_bytes.b1*
define *c2* ≡ *cur_bytes.b2*
define *c3* ≡ *cur_bytes.b3*

⟨ Globals in the outer block 5 ⟩ +≡
cur_bytes: *four_bytes*; { a four-byte accumulator }
zero_bytes: *four_bytes*; { four bytes all zero }

68. ⟨ Set initial values 6 ⟩ +≡
zero_bytes.b0 ← 0; *zero_bytes.b1* ← 0; *zero_bytes.b2* ← 0; *zero_bytes.b3* ← 0;

69. Since the *get_four_bytes* routine is used very infrequently, no attempt has been made to make it fast; we only want it to work.

```
procedure get_four_bytes; { scans an unsigned constant and sets four_bytes }
  var c: integer; { local two-byte accumulator }
    r: integer; { radix }
  begin repeat get_next;
  until cur_char ≠ "␣"; { skip the blanks before the type code }
  r ← 0; cur_bytes ← zero_bytes; { start with the accumulator zero }
  if cur_char = "H" then r ← 16
  else if cur_char = "O" then r ← 8
    else if cur_char = "D" then r ← 10
      else skip_error('Decimal␣("D"),␣octal␣("O"),␣or␣hex␣("H")␣value␣needed␣here');
  if r > 0 then
    begin repeat get_next;
    until cur_char ≠ "␣"; { skip the blanks after the type code }
    while ((cur_char ≥ "0") ∧ (cur_char ≤ "9")) ∨ ((cur_char ≥ "A") ∧ (cur_char ≤ "F")) do
      ⟨ Multiply by r, add cur_char − "0", and get_next 70 ⟩;
    end;
  end;
```

70. ⟨ Multiply by *r*, add *cur_char* − "0", and *get_next* 70 ⟩ ≡

```
  begin if cur_char ≥ "A" then cur_char ← cur_char + "0" + 10 − "A";
  if cur_char ≥ "0" + r then skip_error('Illegal␣digit')
  else begin c ← c3 * r + cur_char − "0"; c3 ← c mod 256;
    c ← c2 * r + c div 256; c2 ← c mod 256;
    c ← c1 * r + c div 256; c1 ← c mod 256;
    c ← c0 * r + c div 256;
    if c < 256 then c0 ← c
    else begin cur_bytes ← zero_bytes;
      if r = 8 then skip_error('Sorry,␣the␣maximum␣octal␣value␣is␣O␣37777777777')
      else if r = 10 then skip_error('Sorry,␣the␣maximum␣decimal␣value␣is␣D␣4294967295')
        else skip_error('Sorry,␣the␣maximum␣hex␣value␣is␣H␣FFFFFFFF');
      end;
    get_next;
    end;
  end
```

This code is used in section 69.

71. The remaining scanning routine is the most interesting. It scans a real constant and returns the nearest *fix_word* approximation to that constant. A *fix_word* is a 32-bit integer that represents a real value that has been multiplied by 2^{20}. Since `VPtoVF` restricts the magnitude of reals to 2048, the *fix_word* will have a magnitude less than 2^{31}.

define *unity* ≡ ´4000000 { 2^{20}, the *fix_word* 1.0 }

⟨ Types in the outer block 23 ⟩ +≡
fix_word = *integer*; { a scaled real value with 20 bits of fraction }

72. When a real value is desired, we might as well treat '`D`' and '`R`' formats as if they were identical.

```
function get_fix: fix_word; { scans a real property value }
  var negative: boolean; { was there a minus sign? }
    acc: integer; { an accumulator }
    int_part: integer; { the integer part }
    j: 0 .. 7; { the number of decimal places stored }
  begin repeat get_next;
  until cur_char ≠ "␣"; { skip the blanks before the type code }
  negative ← false; acc ← 0; { start with the accumulators zero }
  if (cur_char ≠ "R") ∧ (cur_char ≠ "D") then skip_error(´An␣"R"␣or␣"D"␣value␣is␣needed␣here´)
  else begin ⟨ Scan the blanks and/or signs after the type code 73 ⟩;
    while (cur_char ≥ "0") ∧ (cur_char ≤ "9") do ⟨ Multiply by 10, add cur_char − "0", and get_next 74 ⟩;
    int_part ← acc; acc ← 0;
    if cur_char = "." then ⟨ Scan the fraction part and put it in acc 76 ⟩;
    if (acc ≥ unity) ∧ (int_part = 2047) then skip_error(´Real␣constants␣must␣be␣less␣than␣2048´)
    else acc ← int_part * unity + acc;
    end;
  if negative then get_fix ← −acc else get_fix ← acc;
  end;
```

73. ⟨ Scan the blanks and/or signs after the type code 73 ⟩ ≡

```
repeat get_next;
  if cur_char = "-" then
    begin cur_char ← "␣"; negative ← true;
    end
  else if cur_char = "+" then cur_char ← "␣";
until cur_char ≠ "␣"
```

This code is used in section 72.

74. ⟨ Multiply by 10, add *cur_char* − "0", and *get_next* 74 ⟩ ≡

```
begin acc ← acc * 10 + cur_char − "0";
if acc ≥ 2048 then
  begin skip_error(´Real␣constants␣must␣be␣less␣than␣2048´); acc ← 0; cur_char ← "␣";
  end
else get_next;
end
```

This code is used in section 72.

75. To scan the fraction $.d_1 d_2 \ldots$, we keep track of up to seven of the digits d_j. A correct result is obtained if we first compute $f' = \lfloor 2^{21}(d_1 \ldots d_j)/10^j \rfloor$, after which $f = \lfloor (f'+1)/2 \rfloor$. It is possible to have $f = 1.0$.

⟨ Globals in the outer block 5 ⟩ +≡
fraction_digits: **array** [1 .. 7] **of** *integer*; { 2^{21} times d_j }

76. ⟨ Scan the fraction part and put it in *acc* 76 ⟩ ≡

```
begin j ← 0; get_next;
while (cur_char ≥ "0") ∧ (cur_char ≤ "9") do
  begin if j < 7 then
    begin incr(j); fraction_digits[j] ← ´10000000 ∗ (cur_char − "0");
    end;
  get_next;
  end;
acc ← 0;
while j > 0 do
  begin acc ← fraction_digits[j] + (acc div 10); decr(j);
  end;
acc ← (acc + 10) div 20;
end
```

This code is used in section 72.

77. Storing the property values. When property values have been found, they are squirreled away in a bunch of arrays. The header information is unpacked into bytes in an array called *header_bytes*. The ligature/kerning program is stored in an array of type *four_bytes*. Another *four_bytes* array holds the specifications of extensible characters. The kerns and parameters are stored in separate arrays of *fix_word* values. Virtual font data goes into an array *vf* of single-byte values.

We maintain information about at most 256 local fonts. (If this is inadequate, several arrays need to be made longer and we need to output font definitions that go beyond *fnt1* and *fnt_def1* in the `VF` file.)

Instead of storing the design size in the header array, we will keep it in a *fix_word* variable until the last minute. The number of units in the design size is also kept in a *fix_word*.

```
⟨ Globals in the outer block 5 ⟩ +≡
header_bytes: array [header_index] of byte; { the header block }
header_ptr: header_index; { the number of header bytes in use }
design_size: fix_word; { the design size }
design_units: fix_word; { reciprocal of the scaling factor }
frozen_du: boolean; { have we used design_units irrevocably? }
seven_bit_safe_flag: boolean; { does the file claim to be seven-bit-safe? }
lig_kern: array [0 .. max_lig_steps] of four_bytes; { the ligature program }
nl: 0 .. 32767; { the number of ligature/kern instructions so far }
min_nl: 0 .. 32767; { the final value of nl must be at least this }
kern: array [0 .. max_kerns] of fix_word; { the distinct kerning amounts }
nk: 0 .. max_kerns; { the number of entries of kern }
exten: array [0 .. 255] of four_bytes; { extensible character specs }
ne: 0 .. 256; { the number of extensible characters }
param: array [1 .. max_param_words] of fix_word; { FONTDIMEN parameters }
np: 0 .. max_param_words; { the largest parameter set nonzero }
check_sum_specified: boolean; { did the user name the check sum? }
bchar: 0 .. 256; { the right boundary character, or 256 if unspecified }
vf: array [0 .. vf_size] of byte; { stored bytes for VF file }
vf_ptr: 0 .. vf_size; { first unused location in vf }
vtitle_start: 0 .. vf_size; { starting location of VTITLE string }
vtitle_length: byte; { length of VTITLE string }
packet_start: array [byte] of 0 .. vf_size; { beginning location of character packet }
packet_length: array [byte] of integer; { length of character packet }
font_ptr: 0 .. 256; { number of distinct local fonts seen }
cur_font: 0 .. 256; { number of the current local font }
fname_start: array [byte] of 0 .. vf_size; { beginning of local font name }
fname_length: array [byte] of byte; { length of local font name }
farea_start: array [byte] of 0 .. vf_size; { beginning of local font area }
farea_length: array [byte] of byte; { length of local font area }
font_checksum: array [byte] of four_bytes; { local font checksum }
font_number: array [0 .. 256] of four_bytes; { local font id number }
font_at: array [byte] of fix_word; { local font "at size" }
font_dsize: array [byte] of fix_word; { local font design size }
```

78. ⟨ Types in the outer block 23 ⟩ +≡

```
header_index = 0 .. max_header_bytes; indx = 0 .. 77777;
```

79. ⟨ Local variables for initialization 25 ⟩ +≡

```
d: header_index; { an index into header_bytes }
```

80. We start by setting up the default values.

```
define check_sum_loc = 0
define design_size_loc = 4
define coding_scheme_loc = 8
define family_loc = coding_scheme_loc + 40
define seven_flag_loc = family_loc + 20
define face_loc = seven_flag_loc + 3
⟨ Set initial values 6 ⟩ +≡
for d ← 0 to 18 * 4 − 1 do header_bytes[d] ← 0;
header_bytes[8] ← 11; header_bytes[9] ← "U"; header_bytes[10] ← "N"; header_bytes[11] ← "S";
header_bytes[12] ← "P"; header_bytes[13] ← "E"; header_bytes[14] ← "C"; header_bytes[15] ← "I";
header_bytes[16] ← "F"; header_bytes[17] ← "I"; header_bytes[18] ← "E"; header_bytes[19] ← "D";
for d ← family_loc to family_loc + 11 do header_bytes[d] ← header_bytes[d − 40];
design_size ← 10 * unity; design_units ← unity; frozen_du ← false; seven_bit_safe_flag ← false;
header_ptr ← 18 * 4; nl ← 0; min_nl ← 0; nk ← 0; ne ← 0; np ← 0;
check_sum_specified ← false; bchar ← 256;
vf_ptr ← 0; vtitle_start ← 0; vtitle_length ← 0; font_ptr ← 0;
for k ← 0 to 255 do packet_start[k] ← vf_size;
for k ← 0 to 127 do packet_length[k] ← 1;
for k ← 128 to 255 do packet_length[k] ← 2;
```

81. Most of the dimensions, however, go into the *memory* array. There are at most 257 widths, 257 heights, 257 depths, and 257 italic corrections, since the value 0 is required but it need not be used. So *memory* has room for 1028 entries, each of which is a *fix_word*. An auxiliary table called *link* is used to link these words together in linear lists, so that sorting and other operations can be done conveniently.

We also add four "list head" words to the *memory* and *link* arrays; these are in locations *width* through *italic*, i.e., 1 through 4. For example, *link*[*height*] points to the smallest element in the sorted list of distinct heights that have appeared so far, and *memory*[*height*] is the number of distinct heights.

```
define mem_size = 1028 + 4  { number of nonzero memory addresses }
⟨ Types in the outer block 23 ⟩ +≡
pointer = 0 .. mem_size;  { an index into memory }
```

82. The arrays *char_wd*, *char_ht*, *char_dp*, and *char_ic* contain pointers to the *memory* array entries where the corresponding dimensions appear. Two other arrays, *char_tag* and *char_remainder*, hold the other information that TFM files pack into a *char_info_word*.

```
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 }
define bchar_label ≡ char_remainder[256]  { beginning of ligature program for left boundary }
⟨ Globals in the outer block 5 ⟩ +≡
memory: array [pointer] of fix_word;  { character dimensions and kerns }
mem_ptr: pointer;  { largest memory word in use }
link: array [pointer] of pointer;  { to make lists of memory items }
char_wd: array [byte] of pointer;  { pointers to the widths }
char_ht: array [byte] of pointer;  { pointers to the heights }
char_dp: array [byte] of pointer;  { pointers to the depths }
char_ic: array [byte] of pointer;  { pointers to italic corrections }
char_tag: array [byte] of no_tag .. ext_tag;  { character tags }
char_remainder: array [0 .. 256] of 0 .. 65535;
        { pointers to ligature labels, next larger characters, or extensible characters }
```

83. ⟨ Local variables for initialization 25 ⟩ +≡
c: *byte*; { runs through all character codes }

84. ⟨ Set initial values 6 ⟩ +≡
```
bchar_label ← ´77777;
for c ← 0 to 255 do
  begin char_wd[c] ← 0; char_ht[c] ← 0; char_dp[c] ← 0; char_ic[c] ← 0;
  char_tag[c] ← no_tag; char_remainder[c] ← 0;
  end;
memory[0] ← ´17777777777; { an "infinite" element at the end of the lists }
memory[width] ← 0; link[width] ← 0; { width list is empty }
memory[height] ← 0; link[height] ← 0; { height list is empty }
memory[depth] ← 0; link[depth] ← 0; { depth list is empty }
memory[italic] ← 0; link[italic] ← 0; { italic list is empty }
mem_ptr ← italic;
```

85. As an example of these data structures, let us consider the simple routine that inserts a potentially new element into one of the dimension lists. The first parameter indicates the list head (i.e., $h = width$ for the width list, etc.); the second parameter is the value that is to be inserted into the list if it is not already present. The procedure returns the value of the location where the dimension appears in *memory*. The fact that *memory*[0] is larger than any legal dimension makes the algorithm particularly short.

We do have to handle two somewhat subtle situations. A width of zero must be put into the list, so that a zero-width character in the font will not appear to be nonexistent (i.e., so that its *char_wd* index will not be zero), but this does not need to be done for heights, depths, or italic corrections. Furthermore, it is necessary to test for memory overflow even though we have provided room for the maximum number of different dimensions in any legal font, since the `VPL` file might foolishly give any number of different sizes to the same character.

```
function sort_in(h : pointer; d : fix_word): pointer; { inserts into list }
  var p: pointer; { the current node of interest }
  begin if (d = 0) ∧ (h ≠ width) then sort_in ← 0
  else begin p ← h;
    while d ≥ memory[link[p]] do p ← link[p];
    if (d = memory[p]) ∧ (p ≠ h) then sort_in ← p
    else if mem_ptr = mem_size then
        begin err_print(´Memory␣overflow:␣more␣than␣1028␣widths,␣etc´);
        print_ln(´Congratulations!␣It´´s␣hard␣to␣make␣this␣error.´); sort_in ← p;
        end
      else begin incr(mem_ptr); memory[mem_ptr] ← d; link[mem_ptr] ← link[p]; link[p] ← mem_ptr;
        incr(memory[h]); sort_in ← mem_ptr;
        end;
    end;
  end;
```

86. When these lists of dimensions are eventually written to the TFM file, we may have to do some rounding of values, because the TFM file allows at most 256 widths, 16 heights, 16 depths, and 64 italic corrections. The following procedure takes a given list head h and a given dimension d, and returns the minimum m such that the elements of the list can be covered by m intervals of width d. It also sets *next_d* to the smallest value $d' > d$ such that the covering found by this procedure would be different. In particular, if $d = 0$ it computes the number of elements of the list, and sets *next_d* to the smallest distance between two list elements. (The covering by intervals of width *next_d* is not guaranteed to have fewer than m elements, but in practice this seems to happen most of the time.)

```
⟨ Globals in the outer block 5 ⟩ +≡
next_d: fix_word;  { the next larger interval that is worth trying }
```

87. Once again we can make good use of the fact that *memory*[0] is "infinite."

```
function min_cover(h : pointer; d : fix_word): integer;
  var p: pointer;  { the current node of interest }
    l: fix_word;  { the least element covered by the current interval }
    m: integer;  { the current size of the cover being generated }
  begin m ← 0; p ← link[h]; next_d ← memory[0];
  while p ≠ 0 do
    begin incr(m); l ← memory[p];
    while memory[link[p]] ≤ l + d do p ← link[p];
    p ← link[p];
    if memory[p] − l < next_d then next_d ← memory[p] − l;
    end;
  min_cover ← m;
  end;
```

88. The following procedure uses *min_cover* to determine the smallest d such that a given list can be covered with at most a given number of intervals.

```
function shorten(h : pointer; m : integer): fix_word;  { finds best way to round }
  var d: fix_word;  { the current trial interval length }
    k: integer;  { the size of a minimum cover }
  begin if memory[h] > m then
    begin excess ← memory[h] − m; k ← min_cover(h, 0); d ← next_d;  { now the answer is at least d }
    repeat d ← d + d; k ← min_cover(h, d);
    until k ≤ m;  { first we ascend rapidly until finding the range }
    d ← d div 2; k ← min_cover(h, d);  { now we run through the feasible steps }
    while k > m do
      begin d ← next_d; k ← min_cover(h, d);
      end;
    shorten ← d;
    end
  else shorten ← 0;
  end;
```

89. When we are nearly ready to output the TFM file, we will set *index*[p] ← k if the dimension in *memory*[p] is being rounded to the kth element of its list.

```
⟨ Globals in the outer block 5 ⟩ +≡
index: array [pointer] of byte;
excess: byte;  { number of words to remove, if list is being shortened }
```

90. Here is the procedure that sets the *index* values. It also shortens the list so that there is only one element per covering interval; the remaining elements are the midpoints of their clusters.

```
procedure set_indices(h : pointer; d : fix_word);  { reduces and indexes a list }
  var p: pointer;  { the current node of interest }
    q: pointer;  { trails one step behind p }
    m: byte;  { index number of nodes in the current interval }
    l: fix_word;  { least value in the current interval }
  begin q ← h; p ← link[q]; m ← 0;
  while p ≠ 0 do
    begin incr(m); l ← memory[p]; index[p] ← m;
    while memory[link[p]] ≤ l + d do
      begin p ← link[p]; index[p] ← m; decr(excess);
      if excess = 0 then d ← 0;
      end;
    link[q] ← p; memory[p] ← l + (memory[p] − l) div 2; q ← p; p ← link[p];
    end;
  memory[h] ← m;
  end;
```

91. The input phase. We're ready now to read and parse the `VPL` file, storing property values as we go.

```
⟨ Globals in the outer block 5 ⟩ +≡
c: byte;  { the current character or byte being processed }
x: fix_word;  { current dimension of interest }
k: integer;  { general-purpose index }
```

92.
```
⟨ Read all the input 92 ⟩ ≡
  cur_char ← "␣";
  repeat while cur_char = "␣" do get_next;
    if cur_char = "(" then ⟨ Read a font property value 94 ⟩
    else if (cur_char = ")") ∧ ¬input_has_ended then
        begin err_print('Extra␣right␣parenthesis'); incr(loc); cur_char ← "␣";
        end
      else if ¬input_has_ended then junk_error;
  until input_has_ended
```
This code is used in section 180.

93. The *junk_error* routine just referred to is called when something appears in the forbidden area between properties of a property list.

```
procedure junk_error;  { gets past no man's land }
  begin err_print('There''s␣junk␣here␣that␣is␣not␣in␣parentheses'); skip_to_paren;
  end;
```

94. For each font property, we are supposed to read the data from the left parenthesis that is the current value of *cur_char* to the right parenthesis that matches it in the input. The main complication is to recover with reasonable grace from various error conditions that might arise.

```
⟨ Read a font property value 94 ⟩ ≡
  begin get_name;
  if cur_code = comment_code then skip_to_end_of_item
  else if cur_code > character_code then
      flush_error('This␣property␣name␣doesn''t␣belong␣on␣the␣outer␣level')
    else begin ⟨ Read the font property value specified by cur_code 95 ⟩;
      finish_the_property;
      end;
  end
```
This code is used in section 92.

95. ⟨ Read the font property value specified by *cur_code* 95 ⟩ ≡

```
case cur_code of
check_sum_code: begin check_sum_specified ← true; read_four_bytes(check_sum_loc);
  end;
design_size_code: ⟨ Read the design size 98 ⟩;
design_units_code: ⟨ Read the design units 99 ⟩;
coding_scheme_code: read_BCPL(coding_scheme_loc, 40);
family_code: read_BCPL(family_loc, 20);
face_code: header_bytes[face_loc] ← get_byte;
seven_bit_safe_flag_code: ⟨ Read the seven-bit-safe flag 100 ⟩;
header_code: ⟨ Read an indexed header word 101 ⟩;
font_dimen_code: ⟨ Read font parameter list 102 ⟩;
lig_table_code: read_lig_kern;
boundary_char_code: bchar ← get_byte;
virtual_title_code: begin vtitle_start ← vf_ptr; copy_to_end_of_item;
  if vf_ptr > vtitle_start + 255 then
    begin err_print('VTITLE␣clipped␣to␣255␣characters'); vtitle_length ← 255;
    end
  else vtitle_length ← vf_ptr − vtitle_start;
  end;
map_font_code: ⟨ Read a local font list 104 ⟩;
character_code: read_char_info;
end
```

This code is used in section 94.

96. The **case** statement just given makes use of three subroutines that we haven't defined yet. The first of these puts a 32-bit octal quantity into four specified bytes of the header block.

```
procedure read_four_bytes(l : header_index);
  begin get_four_bytes; header_bytes[l] ← c0; header_bytes[l + 1] ← c1; header_bytes[l + 2] ← c2;
  header_bytes[l + 3] ← c3;
  end;
```

97. The second little procedure is used to scan a string and to store it in the "BCPL format" required by TFM files. The string is supposed to contain at most n bytes, including the first byte (which holds the length of the rest of the string).

```
procedure read_BCPL(l : header_index; n : byte);
  var k: header_index;
  begin k ← l;
  while cur_char = "␣" do get_next;
  while (cur_char ≠ "(") ∧ (cur_char ≠ ")") do
    begin if k < l + n then incr(k);
    if k < l + n then header_bytes[k] ← cur_char;
    get_next;
    end;
  if k = l + n then
    begin err_print(´String␣is␣too␣long;␣its␣first␣´, n − 1 : 1, ´␣characters␣will␣be␣kept´);
    decr(k);
    end;
  header_bytes[l] ← k − l;
  while k < l + n − 1 do  { tidy up the remaining bytes by setting them to nulls }
    begin incr(k); header_bytes[k] ← 0;
    end;
  end;
```

98. ⟨ Read the design size 98 ⟩ ≡

```
begin next_d ← get_fix;
if next_d < unity then err_print(´The␣design␣size␣must␣be␣at␣least␣1´)
else design_size ← next_d;
end
```

This code is used in section 95.

99. ⟨ Read the design units 99 ⟩ ≡

```
begin next_d ← get_fix;
if next_d ≤ 0 then err_print(´The␣number␣of␣units␣per␣design␣size␣must␣be␣positive´)
else if frozen_du then err_print(´Sorry,␣it´´s␣too␣late␣to␣change␣the␣design␣units´)
  else design_units ← next_d;
end
```

This code is used in section 95.

100. ⟨ Read the seven-bit-safe flag 100 ⟩ ≡

```
begin while cur_char = "␣" do get_next;
if cur_char = "T" then seven_bit_safe_flag ← true
else if cur_char = "F" then seven_bit_safe_flag ← false
  else err_print(´The␣flag␣value␣should␣be␣"TRUE"␣or␣"FALSE"´);
skip_to_paren;
end
```

This code is used in section 95.

101. ⟨ Read an indexed header word 101 ⟩ ≡

```
begin c ← get_byte;
if c < 18 then skip_error(´HEADER␣indices␣should␣be␣18␣or␣more´)
else if 4 * c + 4 > max_header_bytes then
    skip_error(´This␣HEADER␣index␣is␣too␣big␣for␣my␣present␣table␣size´)
  else begin while header_ptr < 4 * c + 4 do
      begin header_bytes[header_ptr] ← 0; incr(header_ptr);
      end;
    read_four_bytes(4 * c);
    end;
end
```

This code is used in section 95.

102. The remaining kinds of font property values that need to be read are those that involve property lists on higher levels. Each of these has a loop similar to the one that was used at level zero. Then we put the right parenthesis back so that '*finish_the_property*' will be happy; there is probably a more elegant way to do this.

```
define finish_inner_property_list ≡
        begin decr(loc); incr(level); cur_char ← ")";
        end
```

⟨ Read font parameter list 102 ⟩ ≡

```
begin while level = 1 do
  begin while cur_char = "␣" do get_next;
  if cur_char = "(" then ⟨ Read a parameter value 103 ⟩
  else if cur_char = ")" then skip_to_end_of_item
    else junk_error;
  end;
finish_inner_property_list;
end
```

This code is used in section 95.

103. ⟨ Read a parameter value 103 ⟩ ≡

```
begin get_name;
if cur_code = comment_code then skip_to_end_of_item
else if (cur_code < parameter_code) ∨ (cur_code ≥ char_wd_code) then
    flush_error(´This␣property␣name␣doesn´´t␣belong␣in␣a␣FONTDIMEN␣list´)
  else begin if cur_code = parameter_code then c ← get_byte
    else c ← cur_code − parameter_code;
    if c = 0 then flush_error(´PARAMETER␣index␣must␣not␣be␣zero´)
    else if c > max_param_words then
        flush_error(´This␣PARAMETER␣index␣is␣too␣big␣for␣my␣present␣table␣size´)
      else begin while np < c do
          begin incr(np); param[np] ← 0;
          end;
        param[c] ← get_fix; finish_the_property;
        end;
    end;
end
```

This code is used in section 102.

104. **define** *numbers_differ* ≡ (*font_number*[*cur_font*].*b3* ≠ *font_number*[*font_ptr*].*b3*) ∨
(*font_number*[*cur_font*].*b2* ≠ *font_number*[*font_ptr*].*b2*) ∨
(*font_number*[*cur_font*].*b1* ≠ *font_number*[*font_ptr*].*b1*) ∨
(*font_number*[*cur_font*].*b0* ≠ *font_number*[*font_ptr*].*b0*)

⟨ Read a local font list 104 ⟩ ≡
```
begin get_four_bytes; font_number[font_ptr] ← cur_bytes; cur_font ← 0;
while numbers_differ do incr(cur_font);
if cur_font = font_ptr then { it's a new font number }
  if font_ptr < 256 then ⟨ Initialize a new local font 105 ⟩
  else err_print('I␣can␣handle␣only␣256␣different␣mapfonts');
if cur_font = font_ptr then skip_to_end_of_item
else while level = 1 do
    begin while cur_char = "␣" do get_next;
    if cur_char = "(" then ⟨ Read a local font property 106 ⟩
    else if cur_char = ")" then skip_to_end_of_item
      else junk_error;
    end;
finish_inner_property_list;
end
```
This code is used in section 95.

105. ⟨ Initialize a new local font 105 ⟩ ≡
```
begin incr(font_ptr); fname_start[cur_font] ← vf_size; fname_length[cur_font] ← 4;  { NULL }
farea_start[cur_font] ← vf_size; farea_length[cur_font] ← 0; font_checksum[cur_font] ← zero_bytes;
font_at[cur_font] ← '4000000;  { denotes design size of this virtual font }
font_dsize[cur_font] ← '50000000;  { the fix_word for 10 }
end
```
This code is used in section 104.

106. ⟨ Read a local font property 106 ⟩ ≡
```
begin get_name;
if cur_code = comment_code then skip_to_end_of_item
else if (cur_code < font_name_code) ∨ (cur_code > font_dsize_code) then
    flush_error('This␣property␣name␣doesn''t␣belong␣in␣a␣MAPFONT␣list')
  else begin case cur_code of
    font_name_code: ⟨ Read a local font name 107 ⟩;
    font_area_code: ⟨ Read a local font area 108 ⟩;
    font_checksum_code: begin get_four_bytes; font_checksum[cur_font] ← cur_bytes;
      end;
    font_at_code: begin frozen_du ← true;
      if design_units = unity then font_at[cur_font] ← get_fix
      else font_at[cur_font] ← round((get_fix / design_units) * 1048576.0);
      end;
    font_dsize_code: font_dsize[cur_font] ← get_fix;
    end;  { there are no other cases }
    finish_the_property;
    end;
end
```
This code is used in section 104.

107. ⟨ Read a local font name 107 ⟩ ≡
```
begin fname_start[cur_font] ← vf_ptr; copy_to_end_of_item;
if vf_ptr > fname_start[cur_font] + 255 then
  begin err_print('FONTNAME clipped to 255 characters'); fname_length[cur_font] ← 255;
  end
else fname_length[cur_font] ← vf_ptr − fname_start[cur_font];
end
```
This code is used in section 106.

108. ⟨ Read a local font area 108 ⟩ ≡
```
begin farea_start[cur_font] ← vf_ptr; copy_to_end_of_item;
if vf_ptr > farea_start[cur_font] + 255 then
  begin err_print('FONTAREA clipped to 255 characters'); farea_length[cur_font] ← 255;
  end
else farea_length[cur_font] ← vf_ptr − farea_start[cur_font];
end
```
This code is used in section 106.

109. ⟨ Read ligature/kern list 109 ⟩ ≡
```
begin lk_step_ended ← false;
while level = 1 do
  begin while cur_char = " " do get_next;
  if cur_char = "(" then ⟨ Read a ligature/kern command 110 ⟩
  else if cur_char = ")" then skip_to_end_of_item
    else junk_error;
  end;
finish_inner_property_list;
end
```
This code is used in section 180.

110. ⟨ Read a ligature/kern command 110 ⟩ ≡
```
begin get_name;
if cur_code = comment_code then skip_to_end_of_item
else if cur_code < label_code then
    flush_error('This property name doesn''t belong in a LIGTABLE list')
  else begin case cur_code of
    label_code: ⟨ Read a label step 112 ⟩;
    stop_code: ⟨ Read a stop step 114 ⟩;
    skip_code: ⟨ Read a skip step 115 ⟩;
    krn_code: ⟨ Read a kerning step 117 ⟩;
    lig_code, lig_code + 1, lig_code + 2, lig_code + 3, lig_code + 5, lig_code + 6, lig_code + 7, lig_code + 11:
        ⟨ Read a ligature step 116 ⟩;
    end; { there are no other cases ≥ label_code }
    finish_the_property;
    end;
end
```
This code is used in section 109.

111. When a character is about to be tagged, we call the following procedure so that an error message is given in case of multiple tags.

```
procedure check_tag(c : byte); { print error if c already tagged }
  begin case char_tag[c] of
  no_tag: do_nothing;
  lig_tag: err_print('This␣character␣already␣appeared␣in␣a␣LIGTABLE␣LABEL');
  list_tag: err_print('This␣character␣already␣has␣a␣NEXTLARGER␣spec');
  ext_tag: err_print('This␣character␣already␣has␣a␣VARCHAR␣spec');
  end;
  end;
```

112. ⟨ Read a label step 112 ⟩ ≡

```
  begin while cur_char = "␣" do get_next;
  if cur_char = "B" then
    begin bchar_label ← nl; skip_to_paren; { LABEL BOUNDARYCHAR }
    end
  else begin backup; c ← get_byte; check_tag(c); char_tag[c] ← lig_tag; char_remainder[c] ← nl;
    end;
  if min_nl ≤ nl then min_nl ← nl + 1;
  lk_step_ended ← false;
  end
```

This code is used in section 110.

113. **define** *stop_flag* = 128 { value indicating 'STOP' in a lig/kern program }
define *kern_flag* = 128 { op code for a kern step }

⟨ Globals in the outer block 5 ⟩ +≡

```
lk_step_ended: boolean; { was the last LIGTABLE property LIG or KRN? }
krn_ptr: 0 .. max_kerns; { an index into kern }
```

114. ⟨ Read a stop step 114 ⟩ ≡

```
  if ¬lk_step_ended then err_print('STOP␣must␣follow␣LIG␣or␣KRN')
  else begin lig_kern[nl − 1].b0 ← stop_flag; lk_step_ended ← false;
    end
```

This code is used in section 110.

115. ⟨ Read a skip step 115 ⟩ ≡

```
  if ¬lk_step_ended then err_print('SKIP␣must␣follow␣LIG␣or␣KRN')
  else begin c ← get_byte;
    if c ≥ 128 then err_print('Maximum␣SKIP␣amount␣is␣127')
    else if nl + c ≥ max_lig_steps then err_print('Sorry,␣LIGTABLE␣too␣long␣for␣me␣to␣handle')
      else begin lig_kern[nl − 1].b0 ← c;
        if min_nl ≤ nl + c then min_nl ← nl + c + 1;
        end;
    lk_step_ended ← false;
    end
```

This code is used in section 110.

116. ⟨ Read a ligature step 116 ⟩ ≡
```
begin lig_kern[nl].b0 ← 0; lig_kern[nl].b2 ← cur_code − lig_code; lig_kern[nl].b1 ← get_byte;
lig_kern[nl].b3 ← get_byte;
if nl ≥ max_lig_steps − 1 then err_print('Sorry, LIGTABLE too long for me to handle')
else incr(nl);
lk_step_ended ← true;
end
```
This code is used in section 110.

117. ⟨ Read a kerning step 117 ⟩ ≡
```
begin lig_kern[nl].b0 ← 0; lig_kern[nl].b1 ← get_byte; kern[nk] ← get_fix; krn_ptr ← 0;
while kern[krn_ptr] ≠ kern[nk] do incr(krn_ptr);
if krn_ptr = nk then
  begin if nk < max_kerns then incr(nk)
  else begin err_print('Sorry, too many different kerns for me to handle'); decr(krn_ptr);
    end;
  end;
lig_kern[nl].b2 ← kern_flag + (krn_ptr div 256); lig_kern[nl].b3 ← krn_ptr mod 256;
if nl ≥ max_lig_steps − 1 then err_print('Sorry, LIGTABLE too long for me to handle')
else incr(nl);
lk_step_ended ← true;
end
```
This code is used in section 110.

118. Finally we come to the part of VPtoVF's input mechanism that is used most, the processing of individual character data.

⟨ Read character info list 118 ⟩ ≡
```
begin c ← get_byte;  { read the character code that is being specified }
⟨ Print c in octal notation 137 ⟩;
while level = 1 do
  begin while cur_char = " " do get_next;
  if cur_char = "(" then ⟨ Read a character property 119 ⟩
  else if cur_char = ")" then skip_to_end_of_item
    else junk_error;
  end;
if char_wd[c] = 0 then char_wd[c] ← sort_in(width, 0);  { legitimatize c }
finish_inner_property_list;
end
```
This code is used in section 180.

119. ⟨ Read a character property 119 ⟩ ≡

```
begin get_name;
if cur_code = comment_code then skip_to_end_of_item
else if (cur_code < char_wd_code) ∨ (cur_code > var_char_code) then
    flush_error('This␣property␣name␣doesn''t␣belong␣in␣a␣CHARACTER␣list')
  else begin case cur_code of
    char_wd_code: char_wd[c] ← sort_in(width, get_fix);
    char_ht_code: char_ht[c] ← sort_in(height, get_fix);
    char_dp_code: char_dp[c] ← sort_in(depth, get_fix);
    char_ic_code: char_ic[c] ← sort_in(italic, get_fix);
    next_larger_code: begin check_tag(c); char_tag[c] ← list_tag; char_remainder[c] ← get_byte;
      end;
    map_code: read_packet(c);
    var_char_code: ⟨ Read an extensible recipe for c 120 ⟩;
    end;
    finish_the_property;
    end;
end
```

This code is used in section 118.

120. ⟨ Read an extensible recipe for c 120 ⟩ ≡

```
begin if ne = 256 then err_print('At␣most␣256␣VARCHAR␣specs␣are␣allowed')
else begin check_tag(c); char_tag[c] ← ext_tag; char_remainder[c] ← ne;
  exten[ne] ← zero_bytes;
  while level = 2 do
    begin while cur_char = "␣" do get_next;
    if cur_char = "(" then ⟨ Read an extensible piece 121 ⟩
    else if cur_char = ")" then skip_to_end_of_item
      else junk_error;
    end;
  incr(ne); finish_inner_property_list;
  end;
end
```

This code is used in section 119.

121. ⟨ Read an extensible piece 121 ⟩ ≡

```
begin get_name;
if cur_code = comment_code then skip_to_end_of_item
else if (cur_code < var_char_code + 1) ∨ (cur_code > var_char_code + 4) then
    flush_error('This␣property␣name␣doesn''t␣belong␣in␣a␣VARCHAR␣list')
  else begin case cur_code − (var_char_code + 1) of
    0: exten[ne].b0 ← get_byte;
    1: exten[ne].b1 ← get_byte;
    2: exten[ne].b2 ← get_byte;
    3: exten[ne].b3 ← get_byte;
    end;
    finish_the_property;
    end;
end
```

This code is used in section 120.

122. Assembling the mappings. Each `MAP` property is a sequence of `DVI` instructions, for which we need to know some of the opcodes.

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 *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 }

123. We keep stacks of movement values, in order to optimize the `DVI` code in simple cases.

⟨ Globals in the outer block 5 ⟩ +≡
hstack: **array** [0 .. *max_stack*] **of** 0 .. 2; { number of known horizontal movements }
vstack: **array** [0 .. *max_stack*] **of** 0 .. 2; { number of known vertical movements }
wstack, *xstack*, *ystack*, *zstack*: **array** [0 .. *max_stack*] **of** *fix_word*;
stack_ptr: 0 .. *max_stack*;

124. The packet is built by straightforward assembly of DVI instructions.

```
⟨ Declare the vf_fix procedure 128 ⟩
procedure read_packet(c : byte);
  var cc: byte;  { character being typeset }
    x: fix_word;  { movement }
    h, v: 0 .. 2;  { top of hstack and vstack }
    special_start: 0 .. vf_size;  { location of xxx1 command }
    k: 0 .. vf_size;  { loop index }
  begin packet_start[c] ← vf_ptr; stack_ptr ← 0; h ← 0; v ← 0; cur_font ← 0;
  while level = 2 do
    begin while cur_char = "␣" do get_next;
    if cur_char = "(" then ⟨ Read and assemble a list of DVI commands 125 ⟩
    else if cur_char = ")" then skip_to_end_of_item
      else junk_error;
    end;
  while stack_ptr > 0 do
    begin err_print('Missing␣POP␣supplied'); vf_store(pop); decr(stack_ptr);
    end;
  packet_length[c] ← vf_ptr − packet_start[c]; finish_inner_property_list;
  end;
```

125. ⟨ Read and assemble a list of DVI commands 125 ⟩ ≡

```
  begin get_name;
  if cur_code = comment_code then skip_to_end_of_item
  else if (cur_code < select_font_code) ∨ (cur_code > special_hex_code) then
      flush_error('This␣property␣name␣doesn''t␣belong␣in␣a␣MAP␣list')
    else begin case cur_code of
      select_font_code: ⟨ Assemble a font selection 126 ⟩;
      set_char_code: ⟨ Assemble a typesetting instruction 127 ⟩;
      set_rule_code: ⟨ Assemble a rulesetting instruction 129 ⟩;
      move_right_code, move_right_code + 1: ⟨ Assemble a horizontal movement 130 ⟩;
      move_down_code, move_down_code + 1: ⟨ Assemble a vertical movement 131 ⟩;
      push_code: ⟨ Assemble a stack push 132 ⟩;
      pop_code: ⟨ Assemble a stack pop 133 ⟩;
      special_code, special_hex_code: ⟨ Assemble a special command 134 ⟩;
      end;
      finish_the_property;
      end;
  end
```

This code is used in section 124.

126. ⟨ Assemble a font selection 126 ⟩ ≡

```
  begin get_four_bytes; font_number[font_ptr] ← cur_bytes; cur_font ← 0;
  while numbers_differ do incr(cur_font);
  if cur_font = font_ptr then err_print('Undefined␣MAPFONT␣cannot␣be␣selected')
  else if cur_font < 64 then vf_store(fnt_num_0 + cur_font)
    else begin vf_store(fnt1); vf_store(cur_font);
      end;
  end
```

This code is used in section 125.

127. ⟨ Assemble a typesetting instruction 127 ⟩ ≡

```
if cur_font = font_ptr then err_print('Character␣cannot␣be␣typeset␣in␣undefined␣font')
else begin cc ← get_byte;
  if cc ≥ 128 then vf_store(set1);
  vf_store(cc);
  end
```

This code is used in section 125.

128. Here's a procedure that converts a *fix_word* to a sequence of DVI bytes.

⟨ Declare the *vf_fix* procedure 128 ⟩ ≡

```
procedure vf_fix(opcode : byte; x : fix_word);
  var negative: boolean; k: 0 .. 4;  { number of bytes to typeset }
    t: integer;  { threshold }
  begin frozen_du ← true;
  if design_units ≠ unity then x ← round((x/design_units) * 1048576.0);
  if x ≥ 0 then negative ← false
  else begin negative ← true; x ← −1 − x; end;
  if opcode = 0 then
    begin k ← 4; t ← '100000000; end
  else begin t ← 127; k ← 1;
    while x > t do
      begin t ← 256 * t + 255; incr(k);
      end;
    vf_store(opcode + k − 1); t ← t div 128 + 1;
    end;
  repeat if negative then
      begin vf_store(255 − (x div t)); negative ← false; x ← (x div t) * t + t − 1 − x;
      end
    else vf_store((x div t) mod 256);
    decr(k); t ← t div 256;
  until k = 0;
  end;
```

This code is used in section 124.

129. ⟨ Assemble a rulesetting instruction 129 ⟩ ≡

```
begin vf_store(set_rule); vf_fix(0, get_fix); vf_fix(0, get_fix);
end
```

This code is used in section 125.

130. ⟨ Assemble a horizontal movement 130 ⟩ ≡

```
begin if cur_code = move_right_code then x ← get_fix else x ← −get_fix;
if h = 0 then
  begin wstack[stack_ptr] ← x; h ← 1; vf_fix(w1, x); end
else if x = wstack[stack_ptr] then vf_store(w0)
  else if h = 1 then
      begin xstack[stack_ptr] ← x; h ← 2; vf_fix(x1, x); end
    else if x = xstack[stack_ptr] then vf_store(x0)
      else vf_fix(right1, x);
end
```

This code is used in section 125.

131. ⟨ Assemble a vertical movement 131 ⟩ ≡
```
begin if cur_code = move_down_code then x ← get_fix else x ← −get_fix;
if v = 0 then
  begin ystack[stack_ptr] ← x; v ← 1; vf_fix(y1, x); end
else if x = ystack[stack_ptr] then vf_store(y0)
  else if v = 1 then
      begin zstack[stack_ptr] ← x; v ← 2; vf_fix(z1, x); end
    else if x = zstack[stack_ptr] then vf_store(z0)
      else vf_fix(down1, x);
end
```
This code is used in section 125.

132. ⟨ Assemble a stack push 132 ⟩ ≡
```
if stack_ptr = max_stack then { too pushy }
  err_print('Don''t␣push␣so␣much---stack␣is␣full!')
else begin vf_store(push); hstack[stack_ptr] ← h; vstack[stack_ptr] ← v; incr(stack_ptr); h ← 0; v ← 0;
  end
```
This code is used in section 125.

133. ⟨ Assemble a stack pop 133 ⟩ ≡
```
if stack_ptr = 0 then err_print('Empty␣stack␣cannot␣be␣popped')
else begin vf_store(pop); decr(stack_ptr); h ← hstack[stack_ptr]; v ← vstack[stack_ptr];
  end
```
This code is used in section 125.

134. ⟨ Assemble a special command 134 ⟩ ≡
```
begin vf_store(xxx1); vf_store(0); { dummy length }
special_start ← vf_ptr;
if cur_code = special_code then copy_to_end_of_item
else begin repeat x ← get_hex;
    if cur_char > ")" then vf_store(x * 16 + get_hex);
  until cur_char ≤ ")";
  end;
if vf_ptr − special_start > 255 then ⟨ Convert xxx1 command to xxx4 135 ⟩
else vf[special_start − 1] ← vf_ptr − special_start;
end
```
This code is used in section 125.

135. ⟨ Convert *xxx1* command to *xxx4* 135 ⟩ ≡
```
if vf_ptr + 3 > vf_size then
  begin err_print('Special␣command␣being␣clipped---no␣room␣left!');
  vf_ptr ← special_start + 255; vf[special_start − 1] ← 255;
  end
else begin for k ← vf_ptr downto special_start do vf[k + 3] ← vf[k];
  x ← vf_ptr − special_start; vf_ptr ← vf_ptr + 3; vf[special_start − 2] ← xxx4;
  vf[special_start − 1] ← x div '100000000; vf[special_start] ← (x div '200000) mod 256;
  vf[special_start + 1] ← (x div '400) mod 256; vf[special_start + 2] ← x mod 256;
  end
```
This code is used in section 134.

136. The input routine is now complete except for the following code, which prints a progress report as the file is being read.

```
procedure print_octal(c : byte);  { prints three octal digits }
  begin print('''', (c div 64) : 1, ((c div 8) mod 8) : 1, (c mod 8) : 1);
  end;
```

137. ⟨ Print c in octal notation 137 ⟩ ≡

```
  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 }
  end
```

This code is used in section 118.

138. The checking and massaging phase. Once the whole VPL file has been read in, we must check it for consistency and correct any errors. This process consists mainly of running through the characters that exist and seeing if they refer to characters that don't exist. We also compute the true value of *seven_unsafe*; we make sure that the charlists and ligature programs contain no loops; and we shorten the lists of widths, heights, depths, and italic corrections, if necessary, to keep from exceeding the required maximum sizes.

```
⟨ Globals in the outer block 5 ⟩ +≡
seven_unsafe: boolean;  { do seven-bit characters generate eight-bit ones? }
```

139.
```
⟨ Correct and check the information 139 ⟩ ≡
  if nl > 0 then ⟨ Make sure the ligature/kerning program ends appropriately 145 ⟩;
  seven_unsafe ← false;
  for c ← 0 to 255 do
    if char_wd[c] ≠ 0 then ⟨ For all characters g generated by c, make sure that char_wd[g] is nonzero,
        and set seven_unsafe if c < 128 ≤ g 140 ⟩;
  if bchar_label < ´77777 then
    begin c ← 256; ⟨ Check ligature program of c 149 ⟩;
    end;
  if seven_bit_safe_flag ∧ seven_unsafe then print_ln(´The␣font␣is␣not␣really␣seven-bit-safe!´);
  ⟨ Check for infinite ligature loops 154 ⟩;
  ⟨ Doublecheck the lig/kern commands and the extensible recipes 155 ⟩;
  for c ← 0 to 255 do ⟨ Make sure that c is not the largest element of a charlist cycle 142 ⟩;
  ⟨ Put the width, height, depth, and italic lists into final form 144 ⟩
```
This code is used in section 180.

140. The checking that we need in several places is accomplished by three macros that are only slightly tricky.

```
define existence_tail(#) ≡
        begin char_wd[g] ← sort_in(width, 0); print(#, ´␣´); print_octal(c);
        print_ln(´␣had␣no␣CHARACTER␣spec.´);
        end;
      end
define check_existence_and_safety(#) ≡
      begin g ← #;
      if (g ≥ 128) ∧ (c < 128) then seven_unsafe ← true;
      if char_wd[g] = 0 then existence_tail
define check_existence(#) ≡
      begin g ← #;
      if char_wd[g] = 0 then existence_tail
```

```
⟨ For all characters g generated by c, make sure that char_wd[g] is nonzero, and set seven_unsafe if
      c < 128 ≤ g 140 ⟩ ≡
  case char_tag[c] of
  no_tag: do_nothing;
  lig_tag: ⟨ Check ligature program of c 149 ⟩;
  list_tag: check_existence_and_safety(char_remainder[c])(´The␣character␣NEXTLARGER␣than´);
  ext_tag: ⟨ Check the pieces of exten[c] 141 ⟩;
  end
```
This code is used in section 139.

141. ⟨ Check the pieces of $exten[c]$ 141 ⟩ ≡
```
begin if exten[char_remainder[c]].b0 > 0 then
  check_existence_and_safety(exten[char_remainder[c]].b0)('TOP piece of character');
if exten[char_remainder[c]].b1 > 0 then
  check_existence_and_safety(exten[char_remainder[c]].b1)('MID piece of character');
if exten[char_remainder[c]].b2 > 0 then
  check_existence_and_safety(exten[char_remainder[c]].b2)('BOT piece of character');
check_existence_and_safety(exten[char_remainder[c]].b3)('REP piece of character');
end
```
This code is used in section 140.

142. ⟨ Make sure that c is not the largest element of a charlist cycle 142 ⟩ ≡
```
if char_tag[c] = list_tag then
  begin g ← char_remainder[c];
  while (g < c) ∧ (char_tag[g] = list_tag) do g ← char_remainder[g];
  if g = c then
    begin char_tag[c] ← no_tag;
    print('A cycle of NEXTLARGER characters has been broken at'); print_octal(c);
    print_ln('.');
    end;
  end
```
This code is used in section 139.

143. ⟨ Globals in the outer block 5 ⟩ +≡
```
delta: fix_word;   { size of the intervals needed for rounding }
```

144.
```
define round_message(#) ≡
    if delta > 0 then
      print_ln('I had to round some', #, 's by', (((delta+1) div 2)/'4000000) : 1 : 7, 'units.')
```
⟨ Put the width, height, depth, and italic lists into final form 144 ⟩ ≡
```
delta ← shorten(width, 255); set_indices(width, delta); round_message('width');
delta ← shorten(height, 15); set_indices(height, delta); round_message('height');
delta ← shorten(depth, 15); set_indices(depth, delta); round_message('depth');
delta ← shorten(italic, 63); set_indices(italic, delta); round_message('italic correction');
```
This code is used in section 139.

145.
```
define clear_lig_kern_entry ≡   { make an unconditional STOP }
    lig_kern[nl].b0 ← 255; lig_kern[nl].b1 ← 0; lig_kern[nl].b2 ← 0; lig_kern[nl].b3 ← 0
```
⟨ Make sure the ligature/kerning program ends appropriately 145 ⟩ ≡
```
begin if bchar_label < '77777 then   { make room for it }
  begin clear_lig_kern_entry; incr(nl);
  end;   { bchar_label will be stored later }
while min_nl > nl do
  begin clear_lig_kern_entry; incr(nl);
  end;
if lig_kern[nl − 1].b0 = 0 then lig_kern[nl − 1].b0 ← stop_flag;
end
```
This code is used in section 139.

146. It's not trivial to check for infinite loops generated by repeated insertion of ligature characters. But fortunately there is a nice algorithm for such testing, copied here from the program `TFtoPL` where it is explained further.

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 }

147. ⟨ Globals in the outer block 5 ⟩ +≡
lig_ptr: 0 .. *max_lig_steps*; { an index into *lig_kern* }
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 }
tt: *indx*; { temporary register }
x_lig_cycle, *y_lig_cycle*: 0 .. 256; { problematic ligature pair }

148. ⟨ Set initial values 6 ⟩ +≡
hash_ptr ← 0; *y_lig_cycle* ← 256;
for *k* ← 0 **to** *hash_size* **do** *hash*[*k*] ← 0;

149. **define** *lig_exam* ≡ *lig_kern*[*lig_ptr*].*b1*
define *lig_gen* ≡ *lig_kern*[*lig_ptr*].*b3*

⟨ Check ligature program of *c* 149 ⟩ ≡
begin *lig_ptr* ← *char_remainder*[*c*];
repeat if *hash_input*(*lig_ptr*, *c*) **then**
begin if *lig_kern*[*lig_ptr*].*b2* < *kern_flag* **then**
begin if *lig_exam* ≠ *bchar* **then** *check_existence*(*lig_exam*)(´LIG␣character␣examined␣by´);
check_existence(*lig_gen*)(´LIG␣character␣generated␣by´);
if *lig_gen* ≥ 128 **then**
if (*c* < 128) ∨ (*c* = 256) **then**
if (*lig_exam* < 128) ∨ (*lig_exam* = *bchar*) **then** *seven_unsafe* ← *true*;
end
else if *lig_exam* ≠ *bchar* **then** *check_existence*(*lig_exam*)(´KRN␣character␣examined␣by´);
end;
if *lig_kern*[*lig_ptr*].*b0* ≥ *stop_flag* **then** *lig_ptr* ← *nl*
else *lig_ptr* ← *lig_ptr* + 1 + *lig_kern*[*lig_ptr*].*b0*;
until *lig_ptr* ≥ *nl*;
end

This code is used in sections 139 and 140.

150. The *hash_input* procedure is copied from `TFtoPL`, but it is made into a boolean function that returns *false* if the ligature command was masked by a previous one.

```
function hash_input(p, c : indx): boolean;
        { enter data for character c and command in location p, unless it isn't new }
  label 30;  { go here for a quick 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
    begin hash_input ← false; goto 30; end;
  ⟨ Compute the command parameters y, cc, and zz 151 ⟩;
  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
        begin hash_input ← false; goto 30;  { unused ligature command }
        end;
      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;
  hash_input ← true;
30: end;
```

151. ⟨ Compute the command parameters y, cc, and zz 151 ⟩ ≡

```
  y ← lig_kern[p].b1; t ← lig_kern[p].b2; cc ← simple; zz ← lig_kern[p].b3;
  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 150.

152. (More good stuff from TFtoPL.)

```
function f(h,x,y:indx): indx; forward;    { compute f for arguments known to be in hash[h] }
function eval(x,y:indx): indx;    { 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;
```

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

```
function f;
  begin case class[h] of
  simple: do_nothing;
  left_z: begin class[h] ← pending; lig_z[h] ← eval(lig_z[h],y); class[h] ← simple;
    end;
  right_z: begin class[h] ← pending; lig_z[h] ← eval(x,lig_z[h]); class[h] ← simple;
    end;
  both_z: begin class[h] ← pending; lig_z[h] ← eval(eval(x,lig_z[h]),y); class[h] ← simple;
    end;
  pending: begin x_lig_cycle ← x; y_lig_cycle ← y; lig_z[h] ← 257; class[h] ← simple;
    end;    { the value 257 will break all cycles, since it's not in hash }
  end;    { there are no other cases }
  f ← lig_z[h];
  end;
```

154. ⟨ Check for infinite ligature loops 154 ⟩ ≡

```
  if hash_ptr<hash_size then
    for hh ← 1 to hash_ptr do
      begin tt ← hash_list[hh];
      if class[tt]>simple then    { make sure f is well defined }
        tt ← f(tt,(hash[tt]-1) div 256,(hash[tt]-1) mod 256);
      end;
  if (hash_ptr=hash_size)∨(y_lig_cycle<256) then
    begin if hash_ptr<hash_size 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('!');
      end
    else print_ln('Sorry,␣I␣haven''t␣room␣for␣so␣many␣ligature/kern␣pairs!');
    print_ln('All␣ligatures␣will␣be␣cleared.');
    for c ← 0 to 255 do
      if char_tag[c]=lig_tag then
        begin char_tag[c] ← no_tag; char_remainder[c] ← 0;
        end;
    nl ← 0; bchar ← 256; bchar_label ← 77777;
    end
```

This code is used in section 139.

155. The lig/kern program may still contain references to nonexistent characters, if parts of that program are never used. Similarly, there may be extensible characters that are never used, because they were overridden by `NEXTLARGER`, say. This would produce an invalid `TFM` file; so we must fix such errors.

```
define double_check_tail(#) ≡
    if char_wd[0] = 0 then char_wd[0] ← sort_in(width, 0);
  print('Unused␣', #, '␣refers␣to␣nonexistent␣character␣'); print_octal(c); print_ln('!');
  end ;
  end
define double_check_lig(#) ≡
  begin c ← lig_kern[lig_ptr].#;
  if char_wd[c] = 0 then
    if c ≠ bchar then
      begin lig_kern[lig_ptr].# ← 0; double_check_tail
define double_check_ext(#) ≡
  begin c ← exten[g].#;
  if c > 0 then
    if char_wd[c] = 0 then
      begin exten[g].# ← 0; double_check_tail
define double_check_rep(#) ≡
  begin c ← exten[g].#;
  if char_wd[c] = 0 then
    begin exten[g].# ← 0; double_check_tail
```

⟨ Doublecheck the lig/kern commands and the extensible recipes 155 ⟩ ≡

```
if nl > 0 then
  for lig_ptr ← 0 to nl − 1 do
    if lig_kern[lig_ptr].b2 < kern_flag then
      begin if lig_kern[lig_ptr].b0 < 255 then
        begin double_check_lig(b1)('LIG␣step'); double_check_lig(b3)('LIG␣step');
        end;
      end
    else double_check_lig(b1)('KRN␣step');
if ne > 0 then
  for g ← 0 to ne − 1 do
    begin double_check_ext(b0)('VARCHAR␣TOP'); double_check_ext(b1)('VARCHAR␣MID');
    double_check_ext(b2)('VARCHAR␣BOT'); double_check_rep(b3)('VARCHAR␣REP');
    end
```

This code is used in section 139.

156. The TFM output phase. Now that we know how to get all of the font data correctly stored in VPtoVF's memory, it only remains to write the answers out.

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

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

157. The general plan for producing TFM files is long but simple:

⟨ Do the TFM output 157 ⟩ ≡
⟨ Compute the twelve subfile sizes 159 ⟩;
⟨ Output the twelve subfile sizes 160 ⟩;
⟨ Output the header block 162 ⟩;
⟨ Output the character info 164 ⟩;
⟨ Output the dimensions themselves 166 ⟩;
⟨ Output the ligature/kern program 171 ⟩;
⟨ Output the extensible character recipes 172 ⟩;
⟨ Output the parameters 173 ⟩

This code is used in section 181.

158. A TFM file begins with 12 numbers that tell how big its subfiles are. We already know most of these numbers; for example, the number of distinct widths is $memory[width] + 1$, where the +1 accounts for the zero width that is always supposed to be present. But we still should compute the beginning and ending character codes (*bc* and *ec*), the number of header words (*lh*), and the total number of words in the TFM file (*lf*).

⟨ Globals in the outer block 5 ⟩ +≡
bc: *byte*; { the smallest character code in the font }
ec: *byte*; { the largest character code in the font }
lh: *byte*; { the number of words in the header block }
lf: 0 .. 32767; { the number of words in the entire TFM file }
not_found: *boolean*; { has a font character been found? }
temp_width: *fix_word*; { width being used to compute a check sum }

159. It might turn out that no characters exist at all. But VPtoVF keeps going and writes the TFM anyway. In this case *ec* will be 0 and *bc* will be 1.

⟨ Compute the twelve subfile sizes 159 ⟩ ≡
$lh \leftarrow header\_ptr$ **div** 4;
$not\_found \leftarrow true$; $bc \leftarrow 0$;
while *not_found* **do**
if $(char\_wd[bc] > 0) \vee (bc = 255)$ **then** $not\_found \leftarrow false$
else *incr*(*bc*);
$not\_found \leftarrow true$; $ec \leftarrow 255$;
while *not_found* **do**
if $(char\_wd[ec] > 0) \vee (ec = 0)$ **then** $not\_found \leftarrow false$
else *decr*(*ec*);
if $bc > ec$ **then** $bc \leftarrow 1$;
incr(*memory*[*width*]); *incr*(*memory*[*height*]); *incr*(*memory*[*depth*]); *incr*(*memory*[*italic*]);
⟨ Compute the ligature/kern program offset 168 ⟩;
$lf \leftarrow 6 + lh + (ec - bc + 1) + memory[width] + memory[height] + memory[depth] + memory[italic] + nl + lk\_offset + nk + ne + np$;

This code is used in section 157.

160. **define** *out_size*(#) ≡ *out*((#) **div** 256); *out*((#) **mod** 256)

⟨ Output the twelve subfile sizes 160 ⟩ ≡
out_size(*lf*); *out_size*(*lh*); *out_size*(*bc*); *out_size*(*ec*); *out_size*(*memory*[*width*]);
out_size(*memory*[*height*]); *out_size*(*memory*[*depth*]); *out_size*(*memory*[*italic*]); *out_size*(*nl* + *lk_offset*);
out_size(*nk*); *out_size*(*ne*); *out_size*(*np*);

This code is used in section 157.

161. The routines that follow need a few temporary variables of different types.

⟨ Globals in the outer block 5 ⟩ +≡
j: 0 .. *max_header_bytes*; { index into *header_bytes* }
p: *pointer*; { index into *memory* }
q: *width* .. *italic*; { runs through the list heads for dimensions }
par_ptr: 0 .. *max_param_words*; { runs through the parameters }

162. The header block follows the subfile sizes. The necessary information all appears in *header_bytes*, except that the design size and the seven-bit-safe flag must still be set.

⟨ Output the header block 162 ⟩ ≡
if ¬*check_sum_specified* **then** ⟨ Compute the check sum 163 ⟩;
header_bytes[*design_size_loc*] ← *design_size* **div** ′100000000; { this works since *design_size* > 0 }
header_bytes[*design_size_loc* + 1] ← (*design_size* **div** ′200000) **mod** 256;
header_bytes[*design_size_loc* + 2] ← (*design_size* **div** 256) **mod** 256;
header_bytes[*design_size_loc* + 3] ← *design_size* **mod** 256;
if ¬*seven_unsafe* **then** *header_bytes*[*seven_flag_loc*] ← 128;
for *j* ← 0 **to** *header_ptr* − 1 **do** *out*(*header_bytes*[*j*]);

This code is used in section 157.

163. ⟨ Compute the check sum 163 ⟩ ≡
begin *c0* ← *bc*; *c1* ← *ec*; *c2* ← *bc*; *c3* ← *ec*;
for *c* ← *bc* **to** *ec* **do**
if *char_wd*[*c*] > 0 **then**
begin *temp_width* ← *memory*[*char_wd*[*c*]];
if *design_units* ≠ *unity* **then** *temp_width* ← *round*((*temp_width*/*design_units*) ∗ 1048576.0);
temp_width ← *temp_width* + (*c* + 4) ∗ ′20000000; { this should be positive }
c0 ← (*c0* + *c0* + *temp_width*) **mod** 255; *c1* ← (*c1* + *c1* + *temp_width*) **mod** 253;
c2 ← (*c2* + *c2* + *temp_width*) **mod** 251; *c3* ← (*c3* + *c3* + *temp_width*) **mod** 247;
end;
header_bytes[*check_sum_loc*] ← *c0*; *header_bytes*[*check_sum_loc* + 1] ← *c1*;
header_bytes[*check_sum_loc* + 2] ← *c2*; *header_bytes*[*check_sum_loc* + 3] ← *c3*;
end

This code is used in section 162.

164. The next block contains packed *char_info*.

⟨ Output the character info 164 ⟩ ≡
index[0] ← 0;
for *c* ← *bc* **to** *ec* **do**
begin *out*(*index*[*char_wd*[*c*]]); *out*(*index*[*char_ht*[*c*]] ∗ 16 + *index*[*char_dp*[*c*]]);
out(*index*[*char_ic*[*c*]] ∗ 4 + *char_tag*[*c*]); *out*(*char_remainder*[*c*]);
end

This code is used in section 157.

165. When a scaled quantity is output, we may need to divide it by *design_units*. The following subroutine takes care of this, using floating point arithmetic only if *design_units* ≠ 1.0.

```
procedure out_scaled(x : fix_word); { outputs a scaled fix_word }
  var n: byte; { the first byte after the sign }
    m: 0 .. 65535; { the two least significant bytes }
  begin if abs(x/design_units) ≥ 16.0 then
    begin print_ln('The␣relative␣dimension␣', x/'4000000 : 1 : 3, '␣is␣too␣large.');
    print('␣␣(Must␣be␣less␣than␣16*designsize');
    if design_units ≠ unity then print('␣=', design_units/'200000 : 1 : 3, '␣designunits');
    print_ln(')'); x ← 0;
    end;
  if design_units ≠ unity then x ← round((x/design_units) * 1048576.0);
  if x < 0 then
    begin out(255); x ← x + '100000000;
    if x ≤ 0 then x ← 1;
    end
  else begin out(0);
    if x ≥ '100000000 then x ← '77777777;
    end;
  n ← x div '200000; m ← x mod '200000; out(n); out(m div 256); out(m mod 256);
  end;
```

166. We have output the packed indices for individual characters. The scaled widths, heights, depths, and italic corrections are next.

```
⟨ Output the dimensions themselves 166 ⟩ ≡
  for q ← width to italic do
    begin out(0); out(0); out(0); out(0); { output the zero word }
    p ← link[q]; { head of list }
    while p > 0 do
      begin out_scaled(memory[p]); p ← link[p];
      end;
    end;
```

This code is used in section 157.

167. One embarrassing problem remains: The ligature/kern program might be very long, but the starting addresses in *char_remainder* can be at most 255. Therefore we need to output some indirect address information; we want to compute *lk_offset* so that addition of *lk_offset* to all remainders makes all but *lk_offset* distinct remainders less than 256.

For this we need a sorted table of all relevant remainders.

```
⟨ Globals in the outer block 5 ⟩ +≡
label_table: array [0 .. 256] of record rr: −1 .. '77777; { sorted label values }
    cc: byte; { associated characters }
    end;
label_ptr: 0 .. 256; { index of highest entry in label_table }
sort_ptr: 0 .. 256; { index into label_table }
lk_offset: 0 .. 256; { smallest offset value that might work }
t: 0 .. '77777; { label value that is being redirected }
extra_loc_needed: boolean; { do we need a special word for bchar? }
```

168. ⟨ Compute the ligature/kern program offset 168 ⟩ ≡
```
⟨ Insert all labels into label_table 169 ⟩;
if bchar < 256 then
  begin extra_loc_needed ← true; lk_offset ← 1;
  end
else begin extra_loc_needed ← false; lk_offset ← 0;
  end;
⟨ Find the minimum lk_offset and adjust all remainders 170 ⟩;
if bchar_label < ´77777 then
  begin lig_kern[nl − 1].b2 ← (bchar_label + lk_offset) div 256;
  lig_kern[nl − 1].b3 ← (bchar_label + lk_offset) mod 256;
  end
```
This code is used in section 159.

169. ⟨ Insert all labels into label_table 169 ⟩ ≡
```
label_ptr ← 0; label_table[0].rr ← −1;  { sentinel }
for c ← bc to ec do
  if char_tag[c] = lig_tag then
    begin sort_ptr ← label_ptr;  { there's a hole at position sort_ptr + 1 }
    while label_table[sort_ptr].rr > char_remainder[c] 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 ← char_remainder[c]; incr(label_ptr);
    end
```
This code is used in section 168.

170. ⟨ Find the minimum lk_offset and adjust all remainders 170 ⟩ ≡
```
begin sort_ptr ← label_ptr;  { the largest unallocated label }
if label_table[sort_ptr].rr + lk_offset > 255 then
  begin lk_offset ← 0; extra_loc_needed ← false;  { location 0 can do double duty }
  repeat char_remainder[label_table[sort_ptr].cc] ← lk_offset;
    while label_table[sort_ptr − 1].rr = label_table[sort_ptr].rr do
      begin decr(sort_ptr); char_remainder[label_table[sort_ptr].cc] ← lk_offset;
      end;
    incr(lk_offset); decr(sort_ptr);
  until lk_offset + label_table[sort_ptr].rr < 256;
        { N.B.: lk_offset = 256 satisfies this when sort_ptr = 0 }
  end;
if lk_offset > 0 then
  while sort_ptr > 0 do
    begin char_remainder[label_table[sort_ptr].cc] ← char_remainder[label_table[sort_ptr].cc] + lk_offset;
    decr(sort_ptr);
    end;
end
```
This code is used in section 168.

171. ⟨ Output the ligature/kern program 171 ⟩ ≡

```
if extra_loc_needed then { lk_offset = 1 }
  begin out(255); out(bchar); out(0); out(0);
  end
else for sort_ptr ← 1 to lk_offset do { output the redirection specs }
    begin t ← label_table[label_ptr].rr;
    if bchar < 256 then
      begin out(255); out(bchar);
      end
    else begin out(254); out(0);
      end;
    out_size(t + lk_offset);
    repeat decr(label_ptr);
    until label_table[label_ptr].rr < t;
    end;
if nl > 0 then
  for lig_ptr ← 0 to nl − 1 do
    begin out(lig_kern[lig_ptr].b0); out(lig_kern[lig_ptr].b1); out(lig_kern[lig_ptr].b2);
    out(lig_kern[lig_ptr].b3);
    end;
if nk > 0 then
  for krn_ptr ← 0 to nk − 1 do out_scaled(kern[krn_ptr])
```

This code is used in section 157.

172. ⟨ Output the extensible character recipes 172 ⟩ ≡

```
if ne > 0 then
  for c ← 0 to ne − 1 do
    begin out(exten[c].b0); out(exten[c].b1); out(exten[c].b2); out(exten[c].b3);
    end;
```

This code is used in section 157.

173. For our grand finale, we wind everything up by outputting the parameters.

⟨ Output the parameters 173 ⟩ ≡

```
for par_ptr ← 1 to np do
  begin if par_ptr = 1 then ⟨ Output the slant (param[1]) without scaling 174 ⟩
  else out_scaled(param[par_ptr]);
  end
```

This code is used in section 157.

174. ⟨ Output the slant (*param*[1]) without scaling 174 ⟩ ≡

```
begin if param[1] < 0 then
  begin param[1] ← param[1] + ´10000000000; out((param[1] div ´100000000) + 256 − 64);
  end
else out(param[1] div ´100000000);
out((param[1] div ´200000) mod 256); out((param[1] div 256) mod 256); out(param[1] mod 256);
end
```

This code is used in section 173.

175. The VF output phase. Output to *vf_file* is considerably simpler.

```
define id_byte = 202  { current version of VF format }
define vout(#) ≡ write(vf_file, #)
⟨ Globals in the outer block 5 ⟩ +≡
vcount: integer;  { number of bytes written to vf_file }
```

176. We need a routine to output integers as four bytes. Negative values will never be less than -2^{24}.

```
procedure vout_int(x : integer);
  begin if x ≥ 0 then vout(x div ´100000000)
  else begin vout(255); x ← x + ´100000000;
    end;
  vout((x div ´200000) mod 256); vout((x div ´400) mod 256); vout(x mod 256);
  end;
```

177. ⟨ Do the VF output 177 ⟩ ≡

```
vout(pre); vout(id_byte); vout(vtitle_length);
for k ← 0 to vtitle_length − 1 do vout(vf[vtitle_start + k]);
for k ← check_sum_loc to design_size_loc + 3 do vout(header_bytes[k]);
vcount ← vtitle_length + 11;
for cur_font ← 0 to font_ptr − 1 do ⟨ Output a local font definition 178 ⟩;
for c ← bc to ec do
  if char_wd[c] > 0 then ⟨ Output a packet for character c 179 ⟩;
repeat vout(post); incr(vcount);
until vcount mod 4 = 0
```

This code is used in section 180.

178. ⟨ Output a local font definition 178 ⟩ ≡

```
begin vout(fnt_def1); vout(cur_font);
vout(font_checksum[cur_font].b0); vout(font_checksum[cur_font].b1); vout(font_checksum[cur_font].b2);
vout(font_checksum[cur_font].b3); vout_int(font_at[cur_font]); vout_int(font_dsize[cur_font]);
vout(farea_length[cur_font]); vout(fname_length[cur_font]);
for k ← 0 to farea_length[cur_font] − 1 do vout(vf[farea_start[cur_font] + k]);
if fname_start[cur_font] = vf_size then
  begin vout("N"); vout("U"); vout("L"); vout("L");
  end
else for k ← 0 to fname_length[cur_font] − 1 do vout(vf[fname_start[cur_font] + k]);
vcount ← vcount + 12 + farea_length[cur_font] + fname_length[cur_font];
end
```

This code is used in section 177.

179. ⟨ Output a packet for character c 179 ⟩ ≡

```
begin x ← memory[char_wd[c]];
if design_units ≠ unity then x ← round((x/design_units) * 1048576.0);
if (packet_length[c] > 241) ∨ (x < 0) ∨ (x ≥ ´100000000) then
  begin vout(242); vout_int(packet_length[c]); vout_int(c); vout_int(x);
  vcount ← vcount + 13 + packet_length[c];
  end
else begin vout(packet_length[c]); vout(c); vout(x div ´200000); vout((x div ´400) mod 256);
  vout(x mod 256); vcount ← vcount + 5 + packet_length[c];
  end;
if packet_start[c] = vf_size then
  begin if c ≥ 128 then vout(set1);
  vout(c);
  end
else for k ← 0 to packet_length[c] − 1 do vout(vf[packet_start[c] + k]);
end
```

This code is used in section 177.

180. 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.

```
procedure param_enter;
  begin ⟨ Enter the parameter names 57 ⟩;
  end;
procedure vpl_enter;
  begin ⟨ Enter all the VPL names 56 ⟩;
  end;
procedure name_enter;  { enter all names and their equivalents }
  begin ⟨ Enter all the PL names and their equivalents, except the parameter names 55 ⟩;
  vpl_enter; param_enter;
  end;
procedure read_lig_kern;
  var krn_ptr: 0 .. max_kerns;  { an index into kern }
    c: byte;  { runs through all character codes }
  begin ⟨ Read ligature/kern list 109 ⟩;
  end;
procedure read_char_info;
  var c: byte;  { the char }
  begin ⟨ Read character info list 118 ⟩;
  end;
procedure read_input;
  var c: byte;  { header or parameter index }
  begin ⟨ Read all the input 92 ⟩;
  end;
procedure corr_and_check;
  var c: 0 .. 256;  { runs through all character codes }
    hh: 0 .. hash_size;  { an index into hash_list }
    lig_ptr: 0 .. max_lig_steps;  { an index into lig_kern }
    g: byte;  { a character generated by the current character c }
  begin ⟨ Correct and check the information 139 ⟩
  end;
procedure vf_output;
  var c: byte;  { runs through all character codes }
    cur_font: 0 .. 256;  { runs through all local fonts }
    k: integer;  { loop index }
  begin ⟨ Do the VF output 177 ⟩;
  end;
```

181. Here is where VPtoVF begins and ends.

```
begin initialize;
name_enter;
read_input; print_ln('.');
corr_and_check;
⟨ Do the TFM output 157 ⟩;
vf_output;
end.
```

182. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make `VPtoVF` 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.

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